

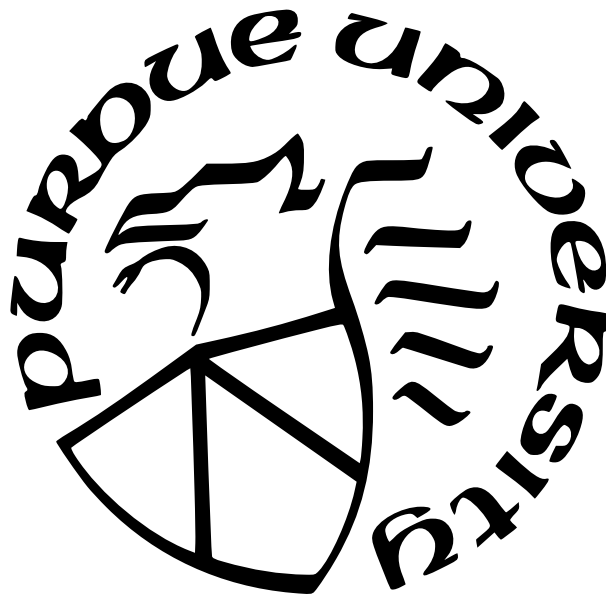
REVAMPING BINARY ANALYSIS WITH SAMPLING AND PROBABILISTIC INFERENCE

by
Zhuo Zhang

A Dissertation

*Submitted to the Faculty of Purdue University
In Partial Fulfillment of the Requirements for the degree of*

Doctor of Philosophy



Department of Computer Science

West Lafayette, Indiana

August 2023

**THE PURDUE UNIVERSITY GRADUATE SCHOOL
STATEMENT OF COMMITTEE APPROVAL**

Dr. Xiangyu Zhang, Chair

Department of Computer Science

Dr. Zeynel B. Celik

Department of Computer Science

Dr. Suresh Jagannathan

Department of Computer Science

Dr. Ninghui Li

Department of Computer Science

Approved by:

Dr. Kihong Park

To my grandfathers, Wen Wang and Xidao Zhang,
who, if they were here, would rejoice in witnessing my accomplishment.

ACKNOWLEDGMENTS

First and foremost, I would like to express my sincere gratitude to my advisor, Dr. Xiangu Zhang, for his unwavering support throughout my Ph.D. journey. His enduring patience, constant motivation, and empathetic understanding over these five years have been a guiding light, helping shape my path as an academic and researcher. Dr. Zhang did not merely teach me; he introduced me to the art and science of research. He showed me the complexities of academic pursuits, taught me how to approach intricate problems with a critical and systematic mindset, and guided me on articulating my ideas with clarity and conciseness. His enthusiasm for innovation and his trust in my abilities have broadened my understanding of computer science and gifted me the freedom to push boundaries and explore new territory. Yet, his guidance was not limited to academics alone. Dr. Zhang also supported me personally, showing me how to make smart plans for the future, make thoughtful decisions, face tough situations bravely, and always aim for self-improvement. I could not have wished for a more exceptional advisor and mentor during my Ph.D. journey.

In addition, I am deeply thankful to my thesis committee members, Dr. Z. Berkay Celik, Dr. Suresh Jagannathan, and Dr. Ninghui Li. Their constructive feedback and encouraging words were essential in refining my dissertation. Their valuable advice on my future career development has left a deep impression on me. The dissertation would not have reached its current level of excellence without their contributions.

I also want to extend my thanks to Dr. Zhiqiang Lin from the Ohio State University. His insightful thinking and words of encouragement empowered me to venture into new research areas previously unknown to me. Dr. Lin's assistance has been instrumental in fostering my confidence to step out of my comfort zone and has motivated me to strive towards novel breakthroughs. His invaluable support has played a crucial role in my academic advancement.

I would also like to express my heartfelt gratitude to my academic comrades, who have greatly enriched my research. In particular, I want to acknowledge Guanhong Tao, my roommate, for his guidance in expanding my understanding of machine learning security, Dr. Guannan Wei for sharing his expertise in programming languages, Dr. Wei You for his valuable lessons on fuzzing and dynamic testing, Le Yu for his wisdom on APT attack

provenance, Yapeng Ye for his guidance on network protocol security, Dr. Wen Xu for his insight into blockchain and smart contract security, Xiangzhe Xu for helping me enhance my knowledge on binary analysis using machine learning approaches, and Wuqi Zhang, for our engaging and insightful discussions about DeFi development and security. Additionally, I extend heartfelt thanks to Siyuan Cheng, Shiwei Feng, Guangyu Shen, and Kaiyuan Zhang. Our countless nights spent together in Room 3133 of Lawson Computer Science Building have not only fostered great camaraderie but have also led to a wealth of shared knowledge. These individuals have significantly shaped my understanding of computer science, and without their support and collaboration, my journey would have been immensely more challenging.

Furthermore, I wish to express my profound gratitude to my collaborators and friends at Purdue University. My sincere thanks to Dr. Yousra Aafer, Shengwei An, Chanwoo Bae, Zhiyuan Cheng, Xuan Chen, Lingyun He, Dr. Yonghwi Kwon, Dr. Wen-chuan Lee, Congyu Liu, Dr. Hongyu Liu, Xuwei Liu, Dr. Yingqi Liu, Yunshu Mao, Dr. Shiqing Ma, Kenneth Miller, Junyang Shao, Dr. Qingkai Shi, Yu Shi, Yi Sun, Zian Su, Fei Wang, Dr. Jianliang Wu, Ruoyu Wu, Yaoxuan Wu, Danning Xie, Zikang Xiong, Qiuling Xu, Zhou Xuan, Lu Yan, Yongwei Yuan, Dr. Juan Zhai, Brian Zhang, Mingwei Zheng, and Jinhao Zhu. Their guidance, friendship, and collaborative spirit have enriched my life over the past five years in West Lafayette.

This dissertation is dedicated to my beloved, Ruixue Zhang, who has been my unfaltering pillar of support and belief throughout not only my Ph.D. journey but also the eight wonderful years we have shared. Even with over seven thousand miles separating us, she has been my rock, my motivation, my beacon in times of despair, and my humbling anchor when pride flooded my judgment. Parallel to supporting me, Ruixue has achieved impressive milestones of her own, notably earning her qualification from the Chinese Institute of Certified Public Accountants over the last five years. Her pursuit of excellence inspires me to become better every day. Her strength, self-motivation, bravery, and optimism have been instrumental in shaping my character. I hold dear the memories of our long FaceTime sessions, where we worked on our own tasks but drew comfort from each other's virtual presence. It is hard to imagine how these past few years would have unfolded without her by my side.

As we continue our journey together, my hope is that we continue to stand by each other, strengthening our bond as a family and jointly navigating the seas of life.

Finally, I wish to express my profound appreciation to my parents, Jianhong Wang and Xuhui Zhang, and all other family members for their unwavering love and immeasurable dedication. Their unconditional support has been my rock. They have always put my needs first, cared for me more than they care for themselves, and have always been there when I needed help. Even though I left my hometown in 2008 and fifteen years have since passed, in their minds, I am still their child, and their desire to protect me remains unwavering. They exhibit an exceptional level of thoughtfulness, always considering my situation. During our phone calls, they worry about taking too much of my time, always concerned that I might be too busy. They even avoid telling me when they are unwell, just so that I would not worry. They have been a steady source of strength for me during the difficult two years of the COVID-19 pandemic. Words fall short in expressing my appreciation for their lifelong support. The regret for not having been by their side over the past fifteen years weighs heavily on me. As I continue my journey, their love and sacrifices will continue to motivate me to be the best I can be.

TABLE OF CONTENTS

LIST OF TABLES	11
LIST OF FIGURES	13
ABSTRACT	16
1 INTRODUCTION	17
1.1 The Thesis	18
1.2 Problem Statement	19
1.3 Overview	21
1.3.1 Program Sampling: Analyzing Data Dependence in Binary Executables	23
1.3.2 Probabilistic Inference: Recovering Variables and Data Structures . .	24
1.3.3 Iterative Refinement: Effective and Efficient Binary-only Fuzzing . .	25
1.3.4 Expanding Viewpoints: Delving into DL-based Binary Analysis . . .	26
1.3.5 Discussion	27
1.4 Contributions and Organization	27
1.5 Publications	30
2 PROGRAM SAMPLING: ANALYZING DATA DEPENDENCE IN BINARY EX- ECUTABLES	33
2.1 Introduction	33
2.2 Motivation	37
2.2.1 Limitations of Existing Techniques	37
2.2.2 Observations	42
2.2.3 Our Technique	42
2.3 Design	45
2.4 Path Sampling	45
2.4.1 Path Counting	46
2.4.2 Path Sampling and Probability Analysis	49
2.4.3 Addressing Practical Challenges	54

2.5	Abstract Interpretation	56
2.6	Posterior Analysis	62
2.7	Evaluation	67
2.7.1	Coverage	69
2.7.2	Program Dependence	70
2.7.3	Applications	77
2.8	Summary	80
3	PROBABILISTIC INFERENCE: RECOVERING VARIABLES AND DATA STRUCTURES	81
3.1	Introduction	81
3.2	Motivation	84
3.2.1	Our Technique	89
3.3	Design Overview	92
3.4	Deterministic Reasoning	92
3.5	Probabilistic Reasoning	99
3.5.1	Probabilistic Inference Rules	100
3.5.2	Probabilistic Constraint Solving	108
3.6	Evaluation	110
3.6.1	Evaluation on Coreutils	111
3.6.2	Evaluation on <i>Howard</i> Benchmark	114
3.6.3	Sensitivity Analysis	116
3.6.4	Execution Time	116
3.6.5	Scalability	117
3.6.6	Impact of Aggressive Optimization	117
3.6.7	Impact of Different Compilers	120
3.6.8	Contribution Breakdown of Different Components	121
3.7	Applications	123
3.7.1	Improving IDA Decompilation	123
3.7.2	Harden Stripped Binary	124

3.8	Summary	127
4	ITERATIVE REFINEMENT: EFFECTIVE AND EFFICIENT BINARY-ONLY FUZZING	128
4.1	Introduction	128
4.2	Motivation	132
4.2.1	Limitations of Existing Technique	132
4.2.2	Our Technique	136
4.3	System Design	141
4.3.1	Probability Analyzer	143
4.3.2	Incremental and Stochastic Rewriting	149
4.3.3	Crash Analyzer	152
4.3.4	Optimizations	153
4.4	Probabilistic Guarantees	154
4.5	Practical Challenges	156
4.6	Evaluation	159
4.6.1	Evaluation on Google FTS	160
4.6.2	Evaluation on Google FTS with Intential Data Inlining	166
4.6.3	Comparison with RetroWrite	170
4.7	Case Studies	171
4.7.1	Finding Zero-days in Closed-source Programs	171
4.7.2	Collect Other Runtime Feedback Than Coverage	172
4.8	Summary	173
5	EXPANDING VIEWPOINTS: DELVING INTO DL-BASED BINARY ANALYSIS	174
5.1	Introduction	174
5.2	Motivation	179
5.3	Design Overview	185
5.4	Syntax-aware Trigger Inversion	186
5.4.1	Trigger Generation	187
5.4.2	Why Not Per-instance Adversarial Attack	189

5.5	Semantic-preserving Trigger Injection	190
5.5.1	Randomized Micro-execution	192
5.5.2	Constraint Generation	196
5.6	Evaluation	201
5.6.1	Experiment Setup	201
5.6.2	Attack Effectiveness	203
5.6.3	Comparison with Baselines	205
5.6.4	Functionality Preservation	213
5.6.5	Why Backdoors Exist in These Models?	213
5.6.6	Runtime Overhead	215
5.6.7	Transfer Attack	217
5.7	Case Study	218
5.8	Summary	220
6	DISCUSSION	221
6.1	Disassembly	221
6.2	Network Protocol Reverse Engineering	223
6.3	Android Security Policy Interpretation	226
6.4	Malware Behavioural Analysis	229
7	RELATED WORK	232
7.1	Program Analysis.	232
7.2	Binary Analysis.	233
7.3	Probabilistic Program Analysis.	234
7.4	N-version Programming.	234
8	CONCLUSION	236
	REFERENCES	238

LIST OF TABLES

2.1	Example of how VSA works on <code>read_words</code>	41
2.2	Interpretation rules of BDA	61
2.3	Example of BDA’s abstract interpretation	62
2.4	Summary of SPECINT 2000 programs	68
2.5	Summary of selected malware samples	68
2.6	Evaluation on memory dependence analysis	71
2.7	Evaluation on the effect of BDA’s posterior analysis and taint tracking	74
2.8	Evaluation on BDA’s runtime overhead	74
2.9	Case study of inferring indirect control flow transfers by BDA	76
2.10	Case study of analyzing malware behaviors by BDA	77
3.1	Evaluation on overall variable recovery (Howard benchmark)	115
3.2	Comparison between OSPREY and Howard	116
3.3	Evaluation on prior probability impact for OSPREY	116
3.4	Evaluation on runtime overhead of variable recovery techniques	117
3.5	Evaluation on optimization impact for OSPREY	118
3.6	Evaluation on effects of BDA and probabilistic inference for OSPREY	121
4.1	Summary of different binary-only fuzzing techniques	133
4.2	Evaluation on soundness of binary-only fuzzing techniques	159
4.3	Evaluation on bug detection capability of binary-only fuzzing techniques	162
4.4	Evaluation on effects of STOCHFuzz’s optimizations	164
4.5	Evaluation on runtime overhead of binary-only fuzzing techniques	165
4.6	Evaluation on fuzzing effectiveness (obfuscated benchmark)	167
4.7	Evaluation on incremental and stochastic rewriting	169
4.8	Evaluation on path coverage achieved by fuzzing (<i>RetroWrite</i> ’s benchmark)	171
4.9	Zero-day vulnerabilities disclosed by STOCHFuzz	171
4.10	Case study of solving mazes by STOCHFuzz (overall results)	172
4.11	Case study of solving mazes by STOCHFuzz (time-to-solve)	173
5.1	Example of PELICAN’s randomized micro-execution	195

5.2	Summary of selected binary analysis models	202
5.3	Evaluation on attack effectiveness of PELICAN	204
5.4	Evaluation on attack success rates of untargeted attacks	210
5.5	Evaluation on runtime overhead of generated binaries	216
5.6	Evaluation on transfer attacks launched by PELICAN	217

LIST OF FIGURES

1.1	Process of compilation and reverse engineering	19
1.2	Process of compilation and reverse engineering	23
2.1	Example to illustrate limitations of existing binary dependency techniques . . .	38
2.2	Examples to illustrate the insights of BDA	43
2.3	CFG of <code>example1</code>	44
2.4	CFG of <code>example3</code>	44
2.5	Architecture of BDA	45
2.6	Weighted CFG of <code>example3</code>	48
2.7	Example of weighted iCFG construction	48
2.8	Weighted iCFG	49
2.9	Example of how BDA handles loops	55
2.10	Language of BDA	57
2.11	Definitions of BDA	59
2.12	Example of BDA's posterior analysis	67
2.13	Evaluation on code coverage achieved by BDA	69
2.14	Evaluation on path coverage achieved by BDA	70
2.15	Evaluation on sampling effect of BDA	72
2.16	Example of BDA's taint tracking	75
2.17	Case study of malware analysis by BDA	78
2.18	Examples of missing and mistyped dependence	79
3.1	Example to illustrate limitations of existing variable recovery techniques	85
3.2	Results of different variable recovery techniques for <code>huft_build</code>	86
3.3	Architecture of OSPREY	91
3.4	Definitions of OSPREY	93
3.5	Primitive analysis facts of OSPREY	93
3.6	Helper functions of OSPREY	94
3.7	Deterministic reasoning rules of OSPREY	95
3.8	Example of OSPREY's deterministic reasoning	98

3.9	Predicate definitions of OSPREY	101
3.10	Example to demonstrate the heap model of OSPREY	102
3.11	Probabilistic inference for primitive and scalar variables in OSPREY	103
3.12	Probabilistic inference for arrays in OSPREY	104
3.13	Probabilistic inference for heap folding in OSPREY	105
3.14	Probabilistic inference for structures in OSPREY	107
3.15	Example of Factor Graph	108
3.16	Example of message passing in Factory Graph	109
3.17	Evaluation on overall variable recovery (recall)	111
3.18	Evaluation on overall variable recovery (precision)	111
3.19	Evaluation on complex variable recovery (recall)	112
3.20	Evaluation on complex variable recovery (precision)	112
3.21	Example of missing data structures by OSPREY	113
3.22	Example of misidentified data structures by OSPREY	113
3.23	Evaluation on tree difference for Coreutils	114
3.24	Evaluation on variable recovery for Apache and Nginx	119
3.25	Evaluation on compiler impact for OSPREY	121
3.26	Case story of decompilation by OSPREY	123
3.27	Case story of decompilation by OSPREY (results)	124
3.28	Case study of ASAN enhancement by OSPREY	125
4.1	Example to illustrate limitations of existing binary-only fuzzing techniques . . .	131
4.2	Example to illustrate limitations of disassembly techniques	135
4.3	Example to illustrate limitations of <i>RetroWrite</i>	136
4.4	Example to illustrate insights of STOCHFuzz	137
4.5	Architecture of STOCHFuzz	142
4.6	Example of Universal Control-flow Graph (UCFG) used by STOCHFuzz	144
4.7	Definitions of STOCHFuzz	145
4.8	Predicates and probabilistic inference rules of OSPREY	146
4.9	Example of factory graphs in STOCHFuzz	151
4.10	Evaluation on fuzzing executions in 24 hours	161

4.11	Evaluation on fuzzing executions in 24 hours (obfuscated benchmark)	168
4.12	Evaluation on the progress of incremental and stochastic rewriting	170
4.13	Evaluation on fuzzing executions in 24 hours (<i>RetroWrite</i> 's benchmark)	170
5.1	Example to illustrate the weakness of DL-based binary analysis models	179
5.2	Pipeline of StateFormer, a DL-based binary analysis model	181
5.3	Example of backdoor generated by NLP trigger inversion techniques	181
5.4	Example of backdoor generated by PELICAN	183
5.5	Architecture of PELICAN	185
5.6	Workflow of PELICAN's syntax-aware trigger inversion	186
5.7	Example of backdoor generated by per-instance adversarial attack	189
5.8	Workflow of PELICAN's semantic-preserving trigger injection	190
5.9	Language of PELICAN	192
5.10	Semantics of PELICAN's randomized micro-execution	194
5.11	Rules of PELICAN's constrain construction	197
5.12	Example of PELICAN's constraint construction	200
5.13	Evaluation on the efficiency of backdoor trigger inversion	205
5.14	Evaluation on attack efficiency	206
5.15	Evaluation on the runtime trigger coverage	207
5.16	Evaluation on attack success rates of targeted attacks	212
5.17	Evaluation on the relation between ASR and the underlying training bias	214
5.18	Case study of black-box attack against DeepDi	219

ABSTRACT

Binary analysis, a cornerstone technique in cybersecurity, enables the examination of binary executables, irrespective of source code availability. It plays a critical role in understanding program behaviors, detecting software bugs, and mitigating potential vulnerabilities, specially in situations where the source code remains out of reach. However, aligning the efficacy of binary analysis with that of source-level analysis remains a significant challenge, primarily due to the uncertainty caused by the loss of semantic information during the compilation process.

This dissertation presents an innovative probabilistic approach, termed as *probabilistic binary analysis*, designed to combat the intrinsic uncertainty in binary analysis. It builds on the fundamental principles of program sampling and probabilistic inference, enhanced further by an iterative refinement architecture. The dissertation suggests that a thorough and practical method of sampling program behaviors can yield a substantial quantity of hints which could be instrumental in recovering lost information, despite the potential inclusion of some inaccuracies. Consequently, a probabilistic inference technique is applied to systematically incorporate and process the collected hints, suppressing the incorrect ones, thereby enabling the interpretation of high-level semantics. Furthermore, an iterative refinement mechanism is deployed to augment the efficiency of the probabilistic analysis in subsequent applications, facilitating the progressive enhancement of analysis outcomes through an automated or human-guided feedback loop.

This work offers an in-depth understanding of the challenges and solutions related to assessing low-level program representations and systematically handling the inherent uncertainty in binary analysis. It aims to contribute to the field by advancing the development of precise, reliable, and interpretable binary analysis solutions, thereby setting the groundwork for future exploration in this domain.

1. INTRODUCTION

Program analysis holds fundamental significance in software security and engineering, facilitating the comprehension of program behaviors, the detection of bugs, and the resolution of potential vulnerabilities, among other essential aspects. Typically, this approach involves examining a program's source code, which usually yields satisfactory results. Nevertheless, in numerous cybersecurity scenarios, access to the source code may be impossible, or relying solely on source code analysis may be inadequate. A few notable instances include:

- **Securing Legacy Software:** Modern computing infrastructures heavily rely on COTS (Commercial Off-the-Shelf) software and legacy software. Much of this software is outdated and potentially prone to security risks. Therefore, understanding the behavior of such legacy software and addressing potential vulnerabilities is vital. However, traditional source code analysis is impractical in these situations, as the software is exclusively distributed in binary format, which is the compiler's output after compiling the source code.
- **Malware Analysis:** Malware comprises malicious software designed to disrupt, damage, or gain unauthorized access to computer systems. Detecting and analyzing malware is one of the most critical tasks in software security. Nonetheless, malware authors often discard the source code and only distribute the binary executable of the malware to evade detection, necessitating analysis of binary executables.
- **Proof-of-Concept (PoC) Development:** A proof-of-concept (PoC) demonstrates the feasibility of a concept, primarily meaning the development of a functional exploit for a vulnerability in the context of software security. PoC development is an essential step in the vulnerability discovery process, as it enables verification of the vulnerability and accelerates its resolution by developers. However, developing an exploit depends on the analysis of low-level machine code and often occurs in situations where programs exhibit undefined behavior (e.g., the vulnerability). In such cases, solely analyzing high-level source code proves to be insufficient and unreliable, as compilers can produce unpredictable machine code in response to undefined behaviors.

The crucial role of machine-code-level analysis in cybersecurity-related tasks has led to the development of *binary analysis*. Such an analysis assesses raw binary executables, aiming at offering performance akin to that of source-level analysis. Nevertheless, binary analysis faces difficulties due to the loss of substantial important semantic information, such as symbol names, data structures, type information, and more, after compilation.

Recovering such information from binary executables is inherently uncertain and can lead to contradictory outcomes of deterministic reasoning. For example, many data structure recovery techniques rely on specific instruction patterns to identify composite data types like `struct { ... }` in C. Unfortunately, these particular instructions might also be present in optimized code snippets that do not access data structures at all. Existing techniques lack a systematic approach for addressing such uncertainty.

In this dissertation, we tackle the intrinsic uncertainty in binary analysis via a novel probabilistic analysis methodology, which is grounded on the principles of *program sampling* and *probabilistic inference*. We note that numerous program behaviors can serve as valuable hints to assist in recovering missing information. However, it is critical to recognize that comprehensively obtaining all such behaviors is infeasible, considering the undecidability of analyzing all non-trivial semantic properties of programs. Despite this challenge, it remains practically feasible to sample these behaviors while tolerating the inclusion of erroneous hints, which we refer to as program sampling. We further propose a probabilistic analysis approach, underpinned by the well-established probabilistic inference technique, to systematically integrate and process the acquired hints, allowing for effective reasoning about missing high-level semantics even in the presence of uncertainty. Additionally, an *iterative refinement* architecture is introduced to enhance the effectiveness of the proposed probabilistic analysis when the downstream application is applicable.

1.1 The Thesis

Program sampling and probabilistic inference can adeptly seize and systematically model the inherent uncertainty present in binary analysis, promoting the development of accurate, robust, and explainable solutions.

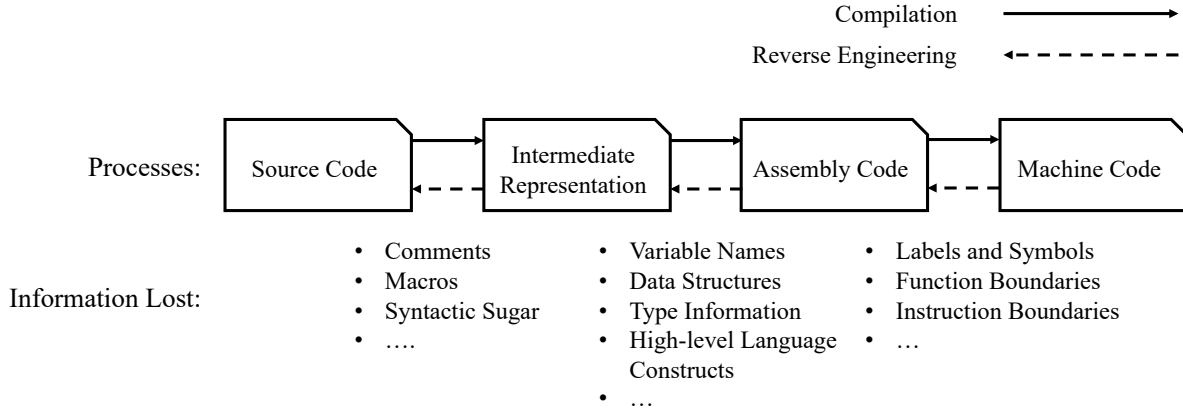


Figure 1.1. The process of compilation and reverse engineering

1.2 Problem Statement

Considering the significance of binary analysis in the field of cybersecurity, it is crucial to elevate the efficacy of binary analysis to a level on par with its source-level counterpart. To this end, binary analysis employs a process inverse to compilation, aiming to reconstruct a high-level representation of a program from a low-level one. This process is often referred to as *reverse engineering*, as illustrated in Figure 1.1. The upper part demonstrates the activities of compilation and reverse engineering, while the lower part showcases the loss of information at each step of compilation. Specifically, the compilation process proceeds from left to right, transforming a high-level program representation (e.g., source code) into a low-level representation (e.g., machine code) through stages of front-end parsing, code generation, and assembly. Conversely, reverse engineering operates in the opposite direction, starting by extracting assembly code from machine code, then recovering intermediate representation from the assembly code, and finally reconstructing source code from the intermediate representation. Note that a substantial amount of information is progressively lost when transitioning from high-level to low-level representations. *This not only complicates the analysis based on low-level representations but also introduces uncertainty into binary analysis.*

To facilitate understanding, we formally define the uncertainty in the context of binary analysis as follows:

Uncertainty. *In the context of binary analysis, uncertainty essentially refers to the possibility of multiple legitimate high-level representations being reconstructed from a single low-level program representation.*

It is important to note that, even when the compilation settings are fixed and predetermined, uncertainty in binary analysis persists, as it is intrinsic to reversing a lossy procedure. For instance, a variable typed as `char` may be compiled into an identical machine code form as a variable typed as `unsigned int8`. As a result, from the perspective of analyzing machine code, it is uncertain which type the variable truly possesses.

In light of these factors, this dissertation seeks to address the following research challenges in binary analysis.

Research Challenge 1. *How can program behavior be adequately comprehended from a low-level representation in a practical manner?*

It is important to note that high-level representations are generally more informative than low-level ones, featuring a more organized structure and richer semantic details (as observed when comparing source code and machine code). Consequently, understanding program behavior from a low-level representation proves more challenging than from a high-level one. For instance, during the data-flow analysis of source code, variable type information often helps reduce the search space of the analysis. However, this is not applicable in binary analysis, where instructions operate solely on raw registers and memory locations without any type information, resulting in an overwhelmingly large search space for the analysis. Existing data-flow analyses for binaries either fail to produce satisfactory results or struggle to scale to complex binaries. \square

Research Challenge 2. *How can the intrinsic uncertainty in binary analysis be systematically modeled and reasoned about?*

The loss of information not only complicates the analysis of low-level representations but also introduces inherent uncertainty into the reverse engineering process. That is to say, even when program behavior can be effectively assessed from a low-level representa-

tion, there remains inevitably uncertainty concerning the information recovered during the reconstruction of high-level representations. For example, in the process of recovering data structures, identifying a data flow between two variables does not necessarily signify that the two variables possess the same type, as possible compilation optimizations might compile various variable types into identical machine code. Binary analysis is conducted in the presence of such uncertainty, which is often overlooked by existing techniques. \square

1.3 Overview

In order to address the challenges outlined earlier, we propose an innovative probabilistic methodology for binary analysis, which is grounded in the following key insights:

- *Program sampling provides a more practical approach for understanding programs in low-level representations, as opposed to conservative analysis.* Recall that, due to uncertainty, multiple legitimate high-level representations can be derived from a specific low-level representation through reverse engineering. A conservative analysis may produce numerous spurious results or fail to scale to complex binaries, since it seeks to encompass all possible high-level representations. However, it is important to note that a given program relation can be revealed by numerous whole-program paths, making a sampling-based approach more practical. This approach samples and analyzes a set of program paths, which is likely sufficient to unveil the program relation of interest. For example, when considering the data dependency relation in a program with n statements, the number of dependencies is $O(n^2)$, while the number of paths could be $O(2^n)$, assuming all branching statements have only two branches. Consequently, a dependence may be exposed by many paths. Sampling a set of paths is likely to reveal all dependencies. We further demonstrate that, if such sampling adheres to a well-designed distribution, the analysis results can be probabilistically guaranteed, meaning the results are likely to be correct with high probability. The details of program sampling are discussed in Chapter 2.
- *Probabilistic inference can naturally model and systematically reason about the intrinsic uncertainty.* Following program sampling, a set of program relations are revealed

and represented as a collection of probabilistic hints. These various hints can be more cohesively integrated using probabilistic inference. Specifically, one can consider each revealed program relation as a piece of evidence, providing a certain degree of confidence in one particular possible high-level representations of all. Although multiple legitimate high-level representations exist due to uncertainty, aggregating all the evidence will likely highlight the most probable one as the correct one. In Chapter 3, we discuss the details of probabilistic inference by solving a concrete problem, specifically, recovering data structures.

Moreover, it is essential to highlight that certain distinct characteristics of cybersecurity applications can potentially facilitate improving the effectiveness and efficacy of binary analysis. In particular, many cybersecurity applications are not a one-time process, but rather entail iterative feedback loops, either automated or manual. For example, during malware analysis, an analyst evaluates the results obtained from the underlying analysis, distinguishes between correct and incorrect outcomes, and submits feedback to improve the analysis. The generation of Proof-of-Concept (PoC) also encompasses a feedback-driven process, where the generated PoC is automatically validated by assessing its capacity to exploit the targeted vulnerability, and the feedback is employed to enhance the PoC generation procedure.

- *The feedback can be seamlessly integrated into the probabilistic inference process as additional hints.* Probabilistic inference naturally enables the integration of new hints into existing analysis results. In particular, when feedback becomes available, it can be translated into fresh probabilistic hints and automatically analyzed by the underlying probabilistic inference procedure. We therefore suggest an iterative refinement architecture, in which the analysis results are progressively refined through feedback. This enhancement is exemplified in Chapter 4, which focuses on binary-only fuzzing.

Figure 1.2 portrays the workflow of the proposed probabilistic binary analysis approach. Specifically, given a machine code of interest, we initially employ program sampling to acquire program behaviors. The probabilistic inference process subsequently leverages these program behaviors to deduce the high-level representation of the program. In cases where downstream

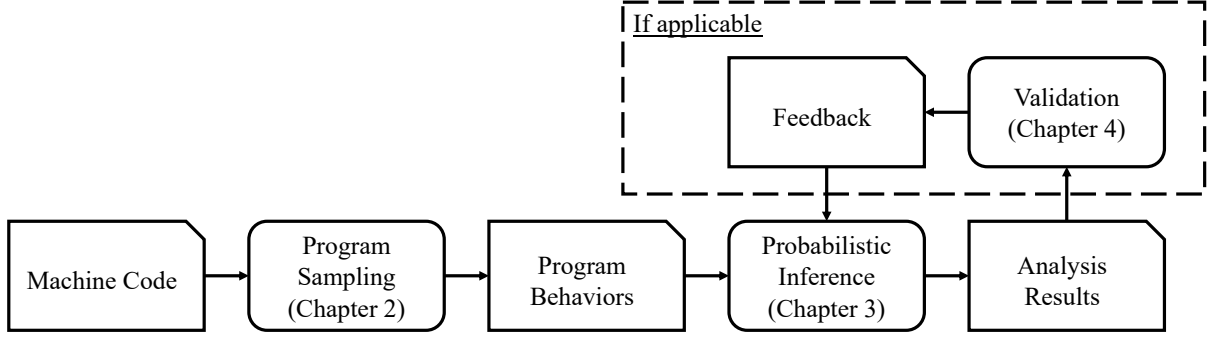


Figure 1.2. The process of compilation and reverse engineering

applications provide feedback, the analysis results can be further refined by incorporating the feedback into the probabilistic inference process.

1.3.1 Program Sampling: Analyzing Data Dependence in Binary Executables

In Chapter 2, we present program sampling, an innovative approach for performing binary analysis. We showcase its potential by using it to examine data dependence in binary executables, given that such analysis is an essential aspect of reverse engineering.

Binary program dependence analysis determines the dependence between two instructions. It is considerably more challenging than source-level dependence analysis, as symbol information (e.g., types and variables) is lost during compilation, and source-level data structures, variables, and arguments are compiled into registers and memory accesses (through registers), making them highly generic and difficult to analyze. Existing works either depends on dynamic dependence analysis, necessitating the availability of high-quality inputs, or conservative static analysis, which compromises scalability and inevitably generates substantial false positives. We note that dependence analysis with probabilistic guarantees may strike an optimal balance between efficacy and practicality. Therefore, we propose a binary level program dependence analysis technique with probabilistic guarantees, enabled by a novel randomized abstract interpretation technique.

Our approach employs program sampling to explore the space of entire program paths in a manner that ensures an equal distribution of probabilities across different paths, irrespective

of path length. Abstract interpretation is further performed on individual sample path, which is different from conservative data-flow analysis that computes/merges the abstract values from all possible paths at each step of interpretation. A context-sensitive and flow-sensitive posterior dependence analysis is conducted to reduce the possible false negatives caused by incomplete path sampling. Probabilistic guarantees can be provided depending on the number of samples taken when certain assumptions are satisfied.

1.3.2 Probabilistic Inference: Recovering Variables and Data Structures

In Chapter 3, we investigate the potential of probabilistic inference in binary analysis by leveraging program behavior insights obtained from program sampling. We demonstrate the effectiveness of this approach in addressing a critical challenge in binary program analysis, namely, the recovery of variable and data structure information. The primary objective is to identify variables, determine their types, and recognize complex array and data structure definitions. Such information is lost during the compilation process, as variables and data structure fields are converted into plain registers and memory locations, devoid of any structural or type information. Access to variables, including simple global scalar variables and complex stack/heap data structure fields with extended reference paths (e.g., a.b.c.d), is uniformly compiled into dereferences of registers. The recovery of this missing information is crucial for ensuring software security.

Traditional techniques rely on a series of hardcoded reverse engineering rules, which are effective under specific conditions (e.g., for binaries generated by particular compilers). However, these techniques often lack general applicability due to the diversity of modern compilers and the prevalence of aggressive optimizations that may disrupt the instruction patterns upon which these rules depend. We note that numerous hints of different types can be gathered to help the recovery of variables and structures; however, these hints have not been fully exploited by existing methods, primarily due to challenges in filtering out spurious hints and the absence of a systematic method for integrating them in the face of uncertainty. To address these shortcomings, we extend our program sampling technique to

collect a comprehensive set of basic behavioral properties of the target binary, including memory access patterns, data-flow characteristics, and points-to relationships.

We introduce random variables to represent the type and structure layout, which are subsequently correlated through the hints gathered by the aforementioned program analysis techniques. Considering the inherent uncertainty of these hints, probabilistic inference is employed to resolve constraints and determine the posterior distribution of the random variables.

1.3.3 Iterative Refinement: Effective and Efficient Binary-only Fuzzing

In Chapter 4, we present an iterative refinement algorithm designed to augment the proposed probabilistic analysis. This architecture is particularly suited for downstream applications that support feedback mechanisms. We illustrate its implementation in the context of binary-only grey-box fuzzing.

Grey-box fuzzing, a prevalent security testing methodology, generates inputs for a target program to identify vulnerabilities. Starting with seed inputs, a fuzzer iteratively executes the program while modifying the inputs. Input mutation is typically guided by coverage information, with popular strategies considering input mutations that improve coverage as significant and subjecting them to further alterations. Consequently, existing fuzzing engines depend on instrumentation to monitor code coverage. They typically employ compilers to facilitate instrumentation prior to fuzzing when source code is accessible. However, in many instances, only binary executables are available.

Several techniques have been devised to enable fuzzing without access to source code, known as binary-only fuzzing. These methods typically involve either resource-intensive dynamic binary rewriting or static rewriting based on restrictive assumptions that are frequently unmet in practice, making binary-only fuzzing a critical and burgeoning field. We observe that fuzzing is a highly iterative process wherein a program is executed repeatedly, affording numerous opportunities for trial-and-error, providing significant amount of feedback, and allowing rewriting to be incremental while improving accuracy over time. Consequently, we propose an innovative incremental and stochastic rewriter that seamlessly integrates with

the fuzzing process. This rewriter employs probabilistic inference to model the uncertainty inherent in addressing the challenges of static binary instrumentation. In essence, our approach does not necessitate sound results from binary analysis as a starting point. Instead, it conducts initial rewriting based on uncertain findings. Through multiple fuzzing iterations, our technique autonomously detects problematic areas, implements repairs, and ultimately achieves effective and efficient binary-only fuzzing.

1.3.4 Expanding Viewpoints: Delving into DL-based Binary Analysis

In Chapter 5, we emphasize the critical role of domain-specific insights and logical reasoning in binary analysis as we venture beyond our initial proposal, investigating the data-driven realm of Deep Learning (DL)-aided binary analysis.

DL has driven remarkable progress in diverse fields such as Computer Vision (CV), Natural Language Processing (NLP), and Robotics. Recent integration of these techniques into binary analysis has yielded promising performance, equating to our probabilistic methods. However, the black-box nature of DL methods casts doubts about their robustness and capacity to generalize. Some studies have raised concerns about DL-based techniques' capacity to effectively process unseen or out-of-distribution (OOD) data, typically referred to as adversarial examples. To investigate whether this potential limitation impacts binary analysis models, we develop an innovative attack strategy against DL-based techniques. This approach uses a trigger inversion method to generate valid instruction sequences and a trigger insertion mechanism to maintain the input binary's semantic integrity. This strategy ensures that the resulting binary retains functional identity with the original while leading the DL-based binary analysis model to erroneous classifications.

Our experimental investigations indicate that DL-based binary analysis techniques are prone to adversarial attacks, underscoring limitations in their ability to generalize. This finding showcases the importance of integrating domain-specific knowledge and logical reasoning when addressing uncertainty in binary analysis, rather than solely relying on data-centric approaches. It also reaffirms the potential of our proposed probabilistic analysis in binary

analysis tasks, suggesting the possible advantages of integrating DL methodologies with our approach.

1.3.5 Discussion

Chapter 6 takes a broader look at the potential uses of our proposed probabilistic analysis in various domains characterized by inherent uncertainties. In this chapter, we focus on how probabilistic analysis can be adapted for an array of tasks that echo binary analysis in the key aspect of deriving high-level abstractions from low-level data. Our observations highlight the adaptability of our probabilistic analysis approach when confronted with uncertainty. We delve into the following problems in detail:

- Disassembly: The challenge of reconstructing assembly code from machine code of various architectures.
- Network Protocol Reverse Engineering: The process of reconstructing protocol specifications from observed network traffic.
- Android Security Policy Interpretation: The demand for recovering security policies and specifications from source code.
- Malware Behavioural Analysis: The task of identifying and understanding malicious behaviors from binary executables.

1.4 Contributions and Organization

The primary contributions of this dissertation include the following:

- We introduce probabilistic binary analysis, an innovative probabilistic methodology for examining binary executables. This approach is underpinned on the concepts of program sampling and probabilistic inference, and can be seamlessly enhanced with available feedback.
- We exhibit the potential of program sampling by utilizing it for binary program dependence analysis. Our technique is facilitated by a novel whole program path sampling

algorithm for comprehensive path exploration, a per-path abstract interpretation approach that is crucial for avoiding spurious abstract values and dependencies, and a posterior analysis to compensate for potential incompleteness in path sampling. We also establish the probabilistic guarantees of our method under certain assumptions. Our evaluation on SPECINT2000 binaries reveals that it scales to intricate binaries, including gcc, while existing techniques struggle to yield results for numerous binaries. In comparison to dynamic dependencies observed during the execution of these binaries with standard inputs, our method misses only 0.19% of dependencies on average. The dependencies reported by our approach are 75 times smaller than those identified by an existing technique.

- We demonstrate how probabilistic inference can benefit programs with inherent uncertainty. Specifically, we propose an innovative probabilistic variable and data structure recovery technique capable of addressing the intrinsic uncertainty of the problem. We develop a set of probabilistic inference rules adept at aggregating profound program behavioral properties to attain precision and extensive coverage in recovery results. We evaluate the performance of our proposed technique against several state-of-the-art methods on two benchmark sets collected from the literature. Our results indicate that our technique surpasses them by 20.41%-56.78% in terms of precision and 11.89%-50.62% in terms of recall. For complex variables (arrays and data structures), our improvement ranges from 6.96%-89.05% (precision) and 46.45%-74.02% (recall).
- We exemplify the seamless integration of feedback into the probabilistic analysis process. Moreover, we propose an innovative incremental and stochastic rewriting technique especially suited for binary-only fuzzing. This method capitalizes on fuzzing and utilizes the numerous fuzzing runs to conduct trial-and-error until precise rewriting is achieved. The technique is supported by a lightweight approach that determines the likelihood of each address representing a data byte, which is formally defined as a probabilistic inference problem. We assess our technique on two standard benchmarks and several commercial binaries. In comparison with state-of-the-art binary-only fuzzers and source-based fuzzers, our results show that our method outperforms binary-only

fuzzers in terms of soundness and efficiency while maintaining comparable performance to source-based fuzzers. For instance, it is 7 times faster than dynamic-based techniques and successfully handles all test programs, whereas other static binary rewriting fuzzers fail on 12.5%-37.5% of the programs. Our fuzzer also identifies zero-days in commercial binaries without any symbol information.

- We explore the critical roles that domain-specific knowledge and logical reasoning play in binary analysis, with a particular emphasis on unearthing backdoor vulnerabilities within deep learning (DL) models used in binary code analysis for security applications. We devise a trigger inversion technique capable of generating valid instructions as backdoor triggers. Additionally, we develop a trigger injection method that ensures the trigger becomes an integral part of the original code’s semantics, and the injected (and patched) code maintains the same semantics as before. This approach features a block-level randomized execution engine and a symbolic patching method. We evaluate our attack on five binary analysis tasks and 15 models. Our assessment reveals that our attack achieves an 86.09% attack success rate (ASR) with only three trigger instructions. We also conduct a case study of exploiting a closed-source commercial tool in a black-box scenario. Our findings underscore the potential of our probabilistic analysis in binary analysis tasks, suggesting potential benefits of integrating DL methodologies with our approach.
- We investigate the potential of applying probabilistic analysis to other tasks that exhibit similarities to binary analysis, specifically in terms of reconstructing high-level abstractions from low-level inputs. We explore several problems, such as disassembly, network protocol reverse engineering, Android security policy analysis, and malware analysis, in detail. Our findings indicate that the probabilistic analysis approach can be broadly adapted when confronted with uncertainty.

This dissertation is organized as follows: Chapter 2 presents BDA, a probabilistic binary dependency analysis technique. Chapter 3 proposes OSPREY, a variable and data structure recovery technique for stripped binaries. Chapter 4 presents STOCHFuzz, an effective and

efficient binary-only fuzzing solution. Chapter 5 delves into the crucial importance of domain-specific knowledge and logical reasoning in binary analysis. Chapter 6 discusses the potential of applying probabilistic analysis to other tasks and domains. Chapter 7 reviews the related works. Finally, Chapter 8 concludes the dissertation.

1.5 Publications

The core research findings of this dissertation are presented in the following publications:

- *BDA: Practical Dependence Analysis for Binary Executables by Unbiased Whole-Program Path Sampling and Per-Path Abstract Interpretation.*

Zhuo Zhang, Wei You, Guanhong Tao, Guannan Wei, Yonghwi Kwon, Xiangyu Zhang.

Proceedings of the ACM on Programming Languages, Volume 3 (OOPSLA 2019).

Presented in Chapter 2.

- *OSPReY: Recovery of Variable and Data Structure via Probabilistic Analysis for Stripped Binary.*

Zhuo Zhang, Yapeng Ye, Wei You, Guanhong Tao, Wen-chuan Lee, Yonghwi Kwon, Yousra Aafer, Xiangyu Zhang.

Proceedings of the 42th IEEE Symposiums on Security and Privacy (S&P 2021).

Presented in Chapter 3.

- *STOCHFuzz: Sound and Cost-effective Fuzzing of Stripped Binaries by Incremental and Stochastic Rewriting.*

Zhuo Zhang, Wei You, Guanhong Tao, Yousra Aafer, Xuwei Liu, Xiangyu Zhang.

Proceedings of the 42th IEEE Symposiums on Security and Privacy (S&P 2021).

Presented in Chapter 4.

- *PELICAN: Exploiting Backdoors of Naturally Trained Deep Learning Models In Binary Code Analysis.*

Zhuo Zhang, Guanhong Tao, Guangyu Shen, Shengwei An, Qiuling Xu, Yingqi Liu, Yapeng Ye, Yaoxuan Wu, Xiangyu Zhang.

Proceedings of the 32nd USENIX Security Symposium (Security 2023).

Presented in Chapter 5.

Additionally, the author has made significant contributions to the following publications related to this dissertation:

- *Probabilistic Disassembly.*

Kenneth Miller, Yonghwi Kwon, Yi Sun, **Zhuo Zhang**, Xiangyu Zhang, Zhiqiang Lin.
Proceedings of the 41st ACM/IEEE International Conference on Software Engineering (ICSE 2019).

Related to Chapter 6.

- *PMP: Cost-Effective Forced Execution with Probabilistic Memory Pre-Planning.*

Wei You, **Zhuo Zhang**, Yonghwi Kwon, Yousra Aafer, Fei Peng, Yu Shi, Carson Makena Harmon, Xiangyu Zhang.

Proceedings of the 41th IEEE Symposiums on Security and Privacy (S&P 2020).

Related to Chapter 6.

- *NetPlier: Probabilistic Network Protocol Reverse Engineering from Message Traces.*

Yapeng Ye, **Zhuo Zhang**, Fei Wang, Xiangyu Zhang, Dongyan Xu.

Proceedings of the 28th Network and Distributed System Security Symposium (NDSS 2021).

Related to Chapter 6.

- *Poirot: Probabilistically Recommending Protections for the Android Framework.*

Zeinab El-Rewini, **Zhuo Zhang**, Yousra Aafer.

Proceedings of the 29th Conference on Computer and Communications Security (CCS 2022).

Related to Chapter 6.

- *D-ARM: Disassembling ARM Binaries by Lightweight Superset Instruction Interpretation and Graph Modeling.*

Yapeng Ye, **Zhuo Zhang**, Qingkai Shi, Yousra Aafer, Xiangyu Zhang.

Proceedings of the 44th IEEE Symposiums on Security and Privacy (S&P 2023).
Related to [Chapter 6](#).

2. PROGRAM SAMPLING: ANALYZING DATA DEPENDENCE IN BINARY EXECUTABLES

Binary program dependence analysis determines dependence between instructions and hence is important for many applications that have to deal with executables without any symbol information. A key challenge is to identify if multiple memory read/write instructions access the same memory location. The state-of-the-art solution is the value set analysis (VSA) that uses abstract interpretation to determine the set of addresses that are possibly accessed by memory instructions. However, VSA is conservative and hence leads to a large number of bogus dependences and then substantial false positives in downstream analyses such as malware behavior analysis. Furthermore, existing public VSA implementations have difficulty scaling to complex binaries. In this chapter, we propose a new binary dependence analysis called BDA enabled by a randomized abstract interpretation technique. It features a novel whole program path sampling algorithm that is not biased by path length, and a per-path abstract interpretation avoiding precision loss caused by merging paths in traditional analyses. It also provides probabilistic guarantees. Our evaluation on SPECINT2000 programs shows that it can handle complex binaries such as `gcc` whereas VSA implementations from the-state-of-art platforms have difficulty producing results for many SPEC binaries. In addition, the dependences reported by BDA are 75 and 6 times smaller than Alto, a scalable binary dependence analysis tool, and VSA, respectively, with only 0.19% of true dependences observed during dynamic execution missed (by BDA). Applying BDA to call graph generation and malware analysis shows that BDA substantially supersedes the commercial tool IDA in recovering indirect call targets and outperforms a state-of-the-art malware analysis tool Cuckoo by disclosing 3 times more hidden payloads.

2.1 Introduction

Binary analysis is a key technique for many applications such as legacy software maintenance [1, 2], reuse [3, 4], hardening [5, 6], debloating [7, 8], commercial-off-the-shelf software security testing [9, 10], malware analysis [11, 12], and reverse engineering (e.g., communi-

cation protocol reverse engineering) [13, 14]. A key binary analysis is program dependence analysis that determines if there is dependence between two instructions. Binary program dependence analysis is much more challenging than source level dependence analysis as symbol information (e.g., types and variables) is lost during compilation and source level data structures, variables, and arguments are compiled down to registers and memory accesses (through registers), which are very generic and difficult to analyze. The analysis is further confounded by indirect control flow (e.g., call instructions with non-constant targets, often induced by virtual methods in object oriented programs), as call targets are difficult to derive statically without type information. The critical challenge in binary dependence analysis is memory alias analysis that determines if memory access instructions may access a same memory location.

Given the importance of binary analysis, there are a number of widely used binary analysis platforms such as IDA [15], CodeSurfer [16], BAP [17], and ANGR [18]. Some of them leverage dynamic dependence analysis, which is highly effective when inputs are available. However, inputs or input specifications are largely lacking in many security applications. While symbolic execution and fuzzing may be used to generate inputs, they have difficulties scaling to lengthy program paths and execution states for complex binaries with complicated input constraints. Therefore, most of these platforms additionally adopt the *Value Set Analysis* (VSA), a static analysis method, to address the memory alias problem (and hence the dependence analysis problem). VSA was proposed by [19]. It computes the set of possible values for the operands of each instruction. Aliases of two memory accesses can be determined by checking if their value sets share common (address) values. VSA uses a *strided interval* to denote a set of values. Each strided interval specifies the lower bound, the upper bound, and the stride. While being compact, strided intervals feature conservativeness. In many cases, they may become simple value ranges (i.e., intervals with stride 1). As such, even though VSA is sound, it has a number of limitations while being used in practice. Specifically, the possible addresses of many memory accesses often degenerate to the entire memory space such that substantial bogus dependences are introduced; when the set of possible address values of a memory write is inflated, the write becomes extremely expensive as it has to update the value set for all the possible addresses. According to our

experiment (see Section 2.7.2), most publicly available implementations of VSA fail to run on many SPECINT2000 programs [20]. In addition, they produce substantial false positives in dependence analysis.

While soundness (i.e., never missing any true positive program dependence) is critical for certain applications such as semantic preserving binary code transformation, for which VSA aims, probabilistic guarantees (i.e., the analysis has a very low likelihood of missing any true positive) are sufficient for many practical applications. For example, a critical and fundamental application of VSA (and dependence analysis) is to derive indirect control flow transfer targets such that precise call graphs can be constructed. The sound and conservative VSA inevitably has a large number of bogus call edges, rendering the resulted call graph not that useful. In contrast, an analysis that can disclose most true (indirect) call edges and have a low chance of missing some may be more useful in practice. Malware behavior analysis [21] aims to understand hidden payloads of malware samples by reporting the system calls performed by the samples and the corresponding concrete arguments of these system calls (e.g., file delete system call with directory argument `"/home"`). Missing a few dependences (by chance) may not critically impact the generated behavior report whereas having a large number of bogus dependences would lead to substantial false positives, significantly enlarging the human inspection efforts. In other applications including functional component identification (for binary debloating) [8], static analysis guided vulnerability detection/fuzzing [9], and protocol reverse engineering [14], dependence analysis with probabilistic guarantees may provide the appropriate trade-offs between effectiveness and practicality.

Therefore in this chapter, we propose a binary level program dependence analysis technique with probabilistic guarantees, enabled by a novel randomized abstract interpretation technique. Specifically, our technique samples the space of whole program paths in a fashion that the likelihood of different paths being taken are evenly distributed, not biased by path length. Note that tossing a fair coin at each conditional statement yields a very biased path distribution such that long paths can hardly be reached. Abstract interpretation is performed on individual sample path, which is different from VSA that operates like a data-flow analysis that computes/merges the abstract values from all possible paths at each step of interpretation. To avoid using value ranges or strided intervals for external inputs, our abstract

interpretation samples input values from pre-defined distribution. Probabilistic guarantees can be provided depending on the number of samples taken when certain assumptions are satisfied. A context-sensitive and flow-sensitive posterior dependence analysis is performed based on the abstract values computed by the large number of sample interpretations. The analysis is able to reduce the possible false negatives caused by incomplete path sampling. It also features strong updates such that false positives can be effectively suppressed.

Our contributions are summarized as follows.

- We propose a novel whole program path sampling algorithm for general path exploration. We also identify the probabilistic guarantees of our sampling algorithm with certain assumptions.
- We devise a per-path abstract interpretation technique that is critical for avoiding bogus abstract values and dependences, and a posterior analysis to compensate the possible incompleteness in path sampling.
- We address a number of practical challenges such as handling loops, recursions, indirect jumps, and indirect calls.
- We propose a new binary program dependence analysis enabled by a novel randomized abstract interpretation technique.
- We develop a prototype BDA [22] and evaluate it on SPECINT2000 binaries. Our evaluation shows that it scales to complex binaries including `gcc`, whereas VSA implementations from popular platforms such as BAP and ANGR fail to produce results for many binaries. When compared to dynamic dependences observed during running these binaries on standard inputs, BDA misses only 0.19% dependences on average. The dependences reported by BDA are 6 times smaller than those by VSA (when it produces results) and 75 times smaller than Alto (another binary dependence analysis tool that scales). We also evaluate BDA in two downstream analysis, one is to identify indirect control flow transfer targets and the other is to study hidden malware behaviors on 12 recent malware samples. In the former analysis, BDA is equally effective as a state-of-the-art commercial tool IDA in identifying indirect jump targets

and substantially outperforms in identifying indirect call targets (4 found by IDA on average versus 767 found by BDA on average). In the malware analysis, BDA substantially outperforms a commercial state-of-the-art malware analysis tool Cuckoo [23] by reporting 3 times more hidden malicious behaviors.

2.2 Motivation

At the binary level, program dependences induced by registers can be easily inferred. The challenge lies in identifying those induced by memory, due to the difficulty of (statically) determining the locations accessed by memory operations. As such, a key challenge in binary dependence analysis, and also in binary analysis in general, is to determine the points-to relations for memory access instructions. In this section, we explain the limitations of existing techniques, present our observations of program dependences (through memory), and motivate the idea of BDA.

2.2.1 Limitations of Existing Techniques

We use 197.parser from the SPEC2000INT benchmark [20] as an example to illustrate the limitations of existing techniques. 197.parser is a word processing program that analyzes the syntactical structure of a given input sentence based on a pre-defined dictionary. Figure 2.1 presents the simplified code of its dictionary initialization logic. In particular, it sets the number of words in the dictionary to a pre-defined value (line 12), reads words from the dictionary file (line 13), and then outputs the longest word (line 14). During the process of reading words, 197.parser maintains a dictionary tree (lines 24) and records the index of the longest word (line 25).

The core of memory alias analysis (and also the downstream dependence analysis) is to statically determine the possible runtime values (PRV) of the address operand of a memory access instruction, which could be a register or a memory location. We call such operands *variables* for easy description. While the problem is undecidable in general, a large collection of approximation algorithms have been proposed to provide various trade-offs between

```

1  #define MAX_LEN 56
2  #define WORD_CNT 1000
3
4  struct Trie{Word *word; Trie *child[26];};
5  struct Word{char val[MAX_LEN]; Trie *node;};
6  struct Dict{long cap; Word words[WORD_CNT];};
7  Dict *dict;
8  Trie *trie;
9
10 void init_dict() {
11     long idx = 0; //index of the longest word
12     dict->cap = WORD_CNT;
13     read_words(dict->words, &idx);
14     output_word(dict->words[idx].val);
15 }
16
17 void read_words(Word* words, long *idx) {
18     int i, j;
19     for (i = 0; i < WORD_CNT; i++) {
20         words[i].node = trie;
21         for (j = 0; j < MAX_LEN; j++) {
22             words[i].val[j] = read_char();
23             /* Do the following things:
24              1. break_if_line_end(words[i].val[j]);
25              2. update_trie(words[i], j);
26              3. update_longest_idx(idx, j);
27             */
28         }
29         words[i].node->word = &(words[i]);
30     }
31 }

```

Figure 2.1. Example to explain the limitations of existing techniques.

efficiency and precision. Among all these efforts, Alto [24] and VSA [19] are two prominent existing efforts. The latter has been the standard for more than a decade.

Alto. Alto abstracts the PRV of a variable as an *address descriptor* $\langle \text{insn}, \text{OFFSET} \rangle$, where *insn* is the instruction that computes a base value and **OFFSET** denotes a set of possible offsets to the base value. For example, assume in line 13 in function `init_dict()` (Figure 2.1), the address of `dict->words` is loaded to register `rdi` by two instructions. The first instruction `i` loads the base `dict` and the second instruction `j` adds the offset of field `words`, which is 8. The address descriptor of `rdi` after `i` and `j` is hence $\langle i, \{0x8\} \rangle$. Alto only models

PRV computation through register operations, not through memory reads and writes. For an instruction `i` that loads a value from a memory location to a register, Alto resets the PRV of the register to a new address descriptor $\langle i, \{0x0\} \rangle$, not being able to inherit the address descriptor stored by the latest memory write to the location. As such, it has to conservatively consider a memory read with a new descriptor can read from any address, and have dependence with any memory write, causing substantial false positives. For example in function `init_dict()`, Alto considers the read of `dict->words` at line 13 is dependent on the write of `dict->cap` at line 12.

VSA. VSA computes PRV by abstract interpretation, modeling operations through both registers and memory. It abstracts the PRV of a variable as a *strided interval* $s[lb, ub]$, where lb and ub specify the lower bound and upper bound of the interval and s is the stride between values in the interval. Intuitively, the strided interval represents the set of integers $\{lb, lb + s, lb + 2s, \dots, ub\}$. Each strided interval may be associated with a memory region, which could be heap (denoted as \mathcal{H}_a where a is the allocation site), stack (denoted as \mathcal{S}_f where f is the corresponding function), or general for non-heap and non-stack values (denoted as \mathcal{G}). There is a special value \top , which indicates all possible values. VSA computes strided intervals following a set of rules. For example, the **addition** rule is defined as follows. Let $SI_1 = s_1[lb_1, ub_1]$ and $SI_2 = s_2[lb_2, ub_2]$ be two strided intervals, and $SI_3 = SI_1 + SI_2$. Then we have the following equation (2.1), with $gcd()$ the greatest common divisor. Observe that the rule is conservative, meaning SI_3 is a super-set of the all the possible sums of the values in SI_1 and SI_2 .

$$SI_3 = gcd(s_1, s_2)[lb_1 + lb_2, ub_1 + ub_2] \quad (2.1)$$

The major limitation of VSA is over-approximation. According to equation (2.1), abstract interpretation may induce bogus PRV at each instruction, due to both the $gcd()$ operation and the simple approximation of lower and upper bounds. Since there are typically a large number of interpretation steps in whole-program analysis, the bogus dependences are aggregated and magnified, making end results not usable. For example, the write of

`words[i].node->word` at line 28 has false dependence with any following memory read according to VSA.

Table 2.1 illustrates how VSA works on the `read_words()` function. Specifically, the strided interval for register `r12` at instruction `b` is `0x0 [0x8, 0x8]`, denoting a constant 8, and the strided interval for register `r14` at instruction `d` is `0x40 [0x0, 0xfa00]`. Intuitively, `r12` corresponds to the `dict->words` variable passed from the `init_dict` function, and `0x8` is the offset of the `words` field in the `Dict` structure. The strided interval of `r14` represents all the possible loop count `i` values at the binary level. Note that the stride is 40, which is the size of `Word`. These two strided intervals are propagated to instruction `g`, where we have the strided interval for `r12+r14` (corresponding to `&(words[i])` at the source level) as `0x40 [0x8, 0xfa08]` according to the addition rule.

The computation of strided intervals is conservative, which may lead to substantial bogus values in PRV. For example, consider the strided interval for `r12+r14+r15` at instruction `m`. The strided interval for `r12+r14` is `0x40 [0x8, 0xfa08]` as mentioned earlier. The strided interval for `r15`, which corresponds to the counter `j` of the inner loop, is `0x1 [0x0, 0x38]`. According to the addition rule, the resulted strided interval is `0x1 [0x8, 0xfa40]`. As we can see that the resulted strided interval is an over-approximation, covering all possible addresses in the memory region of `dict->words`, while only the addresses corresponding to `dict->words[i].vals[j]` should be included. As such, when instruction `m` writes a value read from input, which is denoted as \top due to the lack of input pre-condition, VSA essentially updates the abstract value for all addresses in `dict->words` to \top . Specifically, `words[i]->node` holds a \top value such that when the later instruction `r` writes a value to `words[i].node->word`, which writes to a field of the memory region denoted by `words[i].node`, VSA conservatively writes the value to the entire address space. As a result, any following memory read would have (bogus) dependence with `r`. Moreover, since VSA needs to update the strided interval for all possible addresses, which could be 2^{64} for the 64-bit system, the analysis becomes extremely time-consuming. According to our experience, such phenomenon happens quite often in practice, substantially hindering the applicability of VSA. In Section 2.7.2, our evaluation shows that the state-of-the-art public VSA implementations fail on many SPEC2000 programs.

Table 2.1. How VSA works on the read_words function

SourceCode	AsmCode	Variable	VSA
void read_words(Word* words, long *idx) {	a. sub rsp, 0x30	rsp ;Stack Pointer	$\mathcal{S}_f : 0x0 [-0x30, -0x30]$
	b. mov r12, rdi	r12 ;words	$\mathcal{H}_\alpha : 0x0 [0x8, 0x8]$
for (int i = 0; i < WORDS_CNT; i++) {	c. xor r14, r14		
	d. cmp r14, 0xfa00	r14 ;i*sizeof(Word)	$\mathcal{G} : 0x40 [0x0, 0xfa00]$
	e. jge u.	N/A	N/A
words[i].node = trie;	f. mov r13, [0x601110]	r13 ;trie	$\mathcal{H}_\beta : 0x0 [0x0, 0x0]$
	g. lea rbx, [r12+r14]	r12+r14 ;&(words[i])	$\mathcal{H}_\alpha : 0x40 [0x8, 0xfa08]$
	h. mov [rbx+0x38], r13	rbx+0x38 ; &(words[i].node)	$\mathcal{H}_\alpha : 0x40 [0x40, 0xfa40]$
for (int j = 0; j < MAX_LEN; j++) {	i. xor r15, r15		
	j. cmp r15, 0x38	r15 ;j	$\mathcal{G} : 0x1 [0x0, 0x38]$
	k. jge p.	N/A	N/A
words[i].val[j] = read_char();	l. call read_char	N/A	N/A
	m. mov [r12+r14+r15], rax	r12+r14+r15 ; &(words[i].val[j])	$\mathcal{H}_\alpha : 0x1 [0x8, 0xfa40]$
		rax ;read_char()	\top
} // Inner Loop	n. inc r15	r15 ;j	$\mathcal{G} : 0x1 [0x0, 0x38]$
	o. jmp j.	N/A	N/A
words[i].node->word = &(words[i]);	p. mov rax, [r12+r14+0x38]	r12+r14+0x38 ; &(words[i].node)	$\mathcal{H}_\alpha : 0x40 [0x40, 0xfa40]$
		rax ;words[i].node	\top
	q. lea rbx, [r12+r14]	rbx ;&(words[i])	$\mathcal{H}_\alpha : 0x40 [0x8, 0xfa08]$
	r. mov [rax], rbx	rax ;words[i].node->word	\top
} // Outer Loop	s. add r14, 0x40	r14 ;i*sizeof(Word)	$\mathcal{G} : 0x40 [0x0, 0xfa00]$
	t. jmp d.	N/A	N/A

2.2.2 Observations

Different analyses entail different kinds of sensitivity. For example, the simplest type inference could be path-insensitive, context-insensitive, and even flow-insensitive. As one of the most complex analyses, dependence analysis is flow-sensitive, context-sensitive, and path-sensitive. However, a key observation is that *a dependence relation, which means dependence through memory in our context, can be disclosed by many whole-program paths*. In other words, even though it is context- and path-sensitive, the level of sensitivity is limited. Intuitively, given a program with n statements, the number of dependences is $O(n^2)$, whereas the number of paths could be $O(2^n)$, assuming all branching statements have only two branches. Hence, a dependence may be exposed by many paths. Consider the code snippet `example1` in Figure 2.2, whose control flow graph is shown in Figure 2.3. There are four possible paths, two of which can expose the dependence between lines 20 and 9 regarding variable `i`. Similarly, two paths can expose the dependence between lines 20 and 9 regarding `j`. Essentially, a dependence is likely exposed if one of its exhibition paths is taken. Program dependences are also input sensitive, meaning that a dependence may or may not be present along a same program path depending on input values. Consider `example2` in Figure 2.2. Variables `i` and `j` denote input and are used as array indices. Note that the code has only one path, and the dependence between lines 25 and 27 may or may not be exercised depending on the values of `i` and `j`. According to [25], *run time values of program variables likely fall into a small range*. In our example, assuming both variables have a uniform distribution in range $[0, c]$, the likelihood of the dependence being exercised is $\frac{1}{c}$. If the path is taken n times with randomly sampled `i` and `j` values, the likelihood becomes $1 - \left(1 - \frac{1}{c}\right)^n$, which is close to 1 when n is large.

2.2.3 Our Technique

We propose a sampling based abstract interpretation technique for dependence analysis. Specifically, following a novel algorithm, BDA samples inter-procedural program paths in a way that the likelihood of different paths being sampled follows a uniform distribution, without being biased by path length. In other words, BDA is able to sample as many unique

```

1  #define MAX_LEN 56
2  char val[MAX_LEN];
3  int i, j, *p, *q;
4
5
6  void example1(
7      int arg0, int arg1
8  ){
9      i = 0; j = 0;
10
11     if (arg0) {
12         p = &i; q = &j;
13     } else {
14         p = &j; q = &i;
15     }
16
17     if (arg1) *p = 1;
18     else *q = 1;
19
20     printf("%d %d\n", i, j);
21 }
22
23 void example2(char arg) {
24     scanf("%d %d\n", &i, &j);
25     val[i] = arg;
26
27     printf("%d\n", val[j]);
28 }
29
30 void example3(int *arg) {
31     scanf("%d %d\n", &i, &j);
32
33     val[i] = 0;
34
35     if (check1(arg)) return;
36     if (check2(arg)) return;
37     if (check3(arg)) return;
38     if (check4(arg)) return;
39     if (check5(arg)) return;
40
41     printf("%d\n", val[j]);
42 }

```

Figure 2.2. Examples to illustrate our observations and our technique

paths as possible given a limited budget. For each sample path, abstract interpretation is performed to compute the possible values for individual instructions. During abstract interpretation, external inputs (e.g., user inputs) are randomly sampled from pre-defined distributions; calling contexts are explicitly denoted as call strings; stack memory is denoted as a stack frame with offset; heap memory is denoted by its allocation site; abstract values are updated based on instruction semantics; memory reads/writes are modeled through an abstract store; and path feasibility is partially modeled (details can be found in Section 2.5). Note that the abstract interpretation in BDA is not based on strided intervals. Instead, it is to-some-extent similar to concrete execution, computing a single abstract value at each instruction instance. The values associated with a static instruction is the union of all the values derived for individual instances of the instruction. In the mean time, it is still quite different from concrete execution, which has extreme difficulty ensuring memory safety when path feasibility is not fully modeled, or concrete external inputs are not available. After aggregating the values derived from individual samples, BDA performs an additional

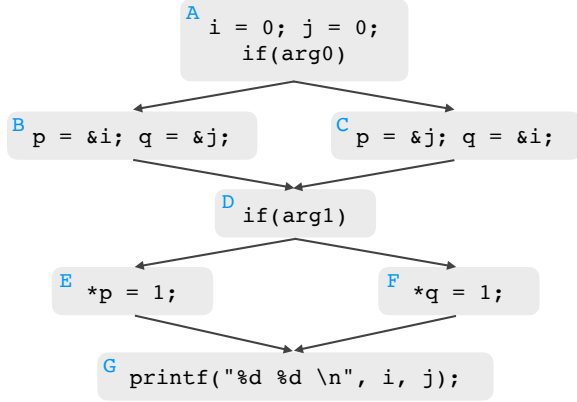


Figure 2.3. CFG of example1

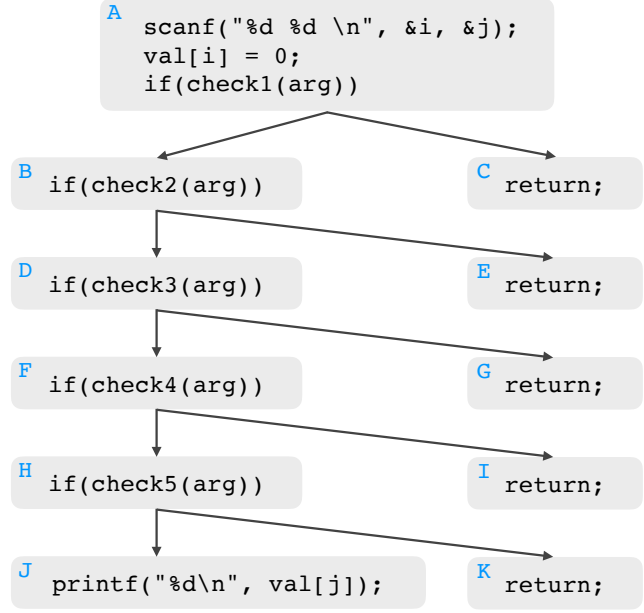


Figure 2.4. CFG of example3

posterior analysis to mitigate the possible incomplete path coverage during sampling. The analysis merges values computed along different branches at each control flow joint point and then cross-checks the address values of memory access instructions to detect dependences. The value merge allows dependences that belong to un-sampled paths to be disclosed with high likelihood.

Note that a naive sampling algorithm that tosses a fair coin at each conditional jump instruction does not work. Consider `example3` in Figure 2.2 with CFG in Figure 2.4. With naive sampling, the path $A \rightarrow C$ gets $\frac{1}{2}$ chance to be taken, while the path $A \rightarrow B \rightarrow D \rightarrow F \rightarrow H \rightarrow J$ has only $\frac{1}{32}$. With the sampling algorithm in BDA, the six paths in the code have an equal chance to be taken. Assuming i and j have the range of $[0, 99]$, BDA guarantees that the dependence between lines 33 and 41 is covered with 99.74% when 60 sample paths are taken. Coming back to our 197.parser example in Figure 2.1, BDA is able to disclose all the true positive dependences in the two functions without generating any false positives.

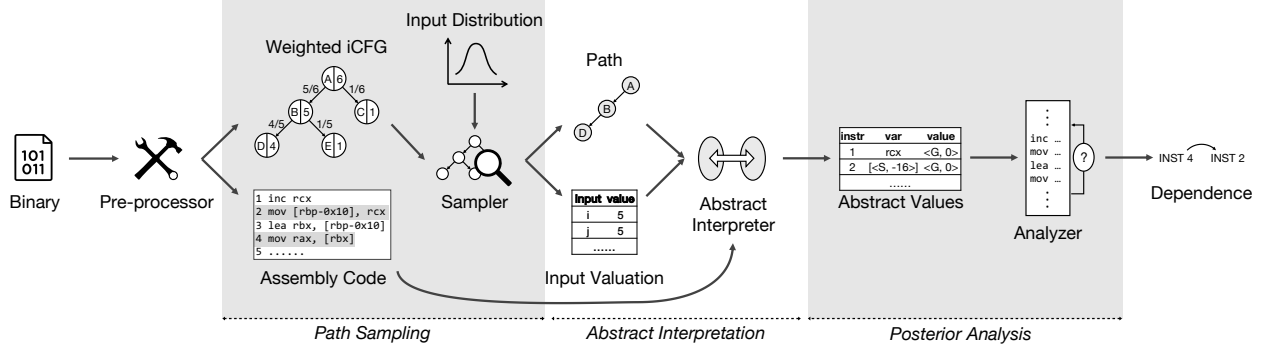


Figure 2.5. Architecture of BDA

2.3 Design

The architecture of BDA is shown in Figure 2.5. It consists of four components: including pre-processor, sampler, abstract interpreter and analyzer. The pre-processor disassembles the given binary to get its assembly code and generates its inter-procedural control flow graph (iCFG) with call edges and return edges explicitly represented. Each basic block of iCFG is weighted by the number of possible inter-procedural paths starting from the block. The sampler samples path based on the weights of blocks and samples external input values based on the pre-defined distributions.

Given a sampled path and input valuation, the abstract interpreter interprets the instructions along the path and computes the abstract values of operands at each instruction. The abstract values for individual instructions are passed to the analyzer for posterior memory dependence analysis. At last, BDA outputs a list of pairs of memory-dependent instructions as analysis results. In the next a few sections, we discuss the details of the individual components.

2.4 Path Sampling

In the sampling step, BDA takes a binary executable and its inter-procedural control flow graph (iCFG), generates a given number of whole-program path samples. Note that we use an iterative method to handle iCFG in the presence of indirect calls, which will be discussed

in Section 2.4.3. The sampling follows a uniform distribution of the space of unique paths. As mentioned in Section 2.2, a simple sampling algorithm that tosses a fair coin at each predicate has strong bias towards short paths.

The basic idea of our sampling algorithm is as follows. For each branching instruction, BDA computes the number of inter-procedural program paths starting from the branch. Sampling bias for the instruction is hence computed from the path counts. Intuitively, a branch leading to more paths has a higher probability to be taken. In order to realize the idea, we address the following two prominent challenges: (1) how to compute the number of inter-procedural paths (in the presence of function calls, loops, and even recursion); and (2) how to sample a strongly-biased distribution as it often occurs that one branch of a conditional statement has a very small number of paths (e.g., those exit upon an error condition) while the other branch has a huge number of paths (e.g., beyond the maximum integer that can be represented in 64 bits). We also study the probabilistic guarantee of our sampling algorithm.

2.4.1 Path Counting

Our path counting algorithm is inspired by the seminal path encoding algorithm in [26]. In Ball-Larus (BL) path encoding, the number of paths starting from a node is the sum of the numbers of paths of its children. It transforms a CFG to its acyclic version (e.g., by removing back-edges) and then computes the path count for each node in a reverse topological order. Figure 2.6 shows the path count for each node (called *node weight* from this point on) for the code in Figure 2.4. Each node is annotated with node id and its weight. Observe that the leaf nodes have weight 1. Then node *H* is computed to have weight 2, *F* has weight 3, and so on. The fractions along edges denote the sampling bias. For example, at node *A*, the chance to take $A \rightarrow B$ is $\frac{5}{6}$ whereas $A \rightarrow C$ is $\frac{1}{6}$. The probabilities of taking the 6 different paths are all $\frac{1}{6}$. However, the BL path counting algorithm is intra-procedural and does not consider loop iterations. Hence, we propose a new whole-program path counting algorithm. To simplify our discussion, we assume the subject program is loop-free and recursion-free,

Algorithm 1 Path Counting

INPUT: $iCFG$	▷ loop-free and recursion-free iCFG of the target binary
OUTPUT: W :	▷ weight (i.e., path count) for each node, a K-bits integer

```
1: function PATHCOUNTING( $iCFG$ )
2:   for  $iaddr$  in reverse topological order of  $iCFG$  do
3:     if  $iaddr$  is a return node then                                ▷ for a return node, initialize its weight to 1
4:        $W[iaddr] \leftarrow 1$ 
5:     else if  $iaddr$  is a call node then                                ▷ function invocation instruction
6:        $callee \leftarrow$  call target of  $iaddr$ 
7:        $ret\_addr \leftarrow$  the instruction right after  $iaddr$ 
8:        $W[iaddr] \leftarrow W[ret\_addr] \times W[callee]$                 ▷ K-digits multiplication, with complexity
                                    $O(K \log K)$ 
9:     else                                                                ▷ other instructions
10:       $W[iaddr] \leftarrow 0$ 
11:      for  $succ$  in successors of  $iaddr$  do
12:         $W[iaddr] \leftarrow W[iaddr] + W[succ]$ 
13:      end for
14:    end if
15:  end for
16:  return  $W$ 
17: end function
```

but has calls and returns. Moreover, each callee must return to its caller and there are no indirect calls. In Section 2.4.3, we will explain how to address these practical issues.

In order to handle inter-procedural path counting, we have to precisely determine the weight (i.e., the number of paths) of an invocation instruction. The key observation is that the weight of an invocation to a callee function `foo()` is the product of the number of inter-procedural paths from the entry of `foo()` to the exit of `foo()`, including paths in the callees of `foo()`, and the weight of the instruction right after the invocation instruction in the caller. The former is called the *callee paths* and the latter is called the *continuation paths*.

The procedure is explained in details in Algorithm 1. It takes the inter-procedural CFG of the binary, and computes the weight for each node, *which denotes the number of inter-procedural paths from the node to the exit of its enclosing function*. Since the input $iCFG$ does not have loops or recursion, we can perform topological sort on the graph. Intuitively, one can consider that we first sort the call graph and then sort the nodes inside each function. The loop in lines 2-15 traverses each node in the reverse topological order. If it is a return instruction, its weight is set to 1 (line 4). If it is a call, the weight is computed as the

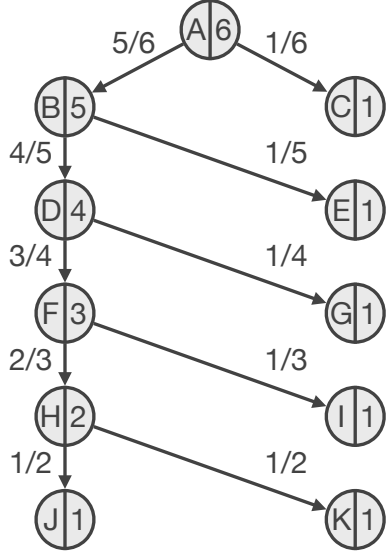


Figure 2.6. Weighted CFG for Fig. 2.4

```

1 void gee(int *a) {
2     if (input()) *a=0;
3     else *a=2;
4 }
5
6 void foo(int *a) {
7     gee(a);
8     if (input()) *a+=1;
9 }
10
11 int main() {
12     int a;
13     if (input()) gee(&a);
14     else foo(&a);
15 }

```

Figure 2.7. Code example with functions

product of the weight of the return address and the weight of the entry point of the callee (i.e., the number of inter-procedural paths inside the callee). Since a method may have a huge number of such paths, which we assume to be bounded by 2^K , the complexity of such product is $O(K \log(K))$. In practice, we find using $K = 600,000$ bits to represent weights is enough. In lines 10-13, if the node is neither call nor return, its weight is the sum of the children weights.

Example. Consider the example in Figure 2.7, which has three functions `main()`, `gee()`, `foo()`, with both `main()` and `foo()` calling `gee()`. The weighted iCFG is shown in Figure 2.8. Following reverse topological order, `gee()` is processed first. As such, $W[A] = 1$ and $W[D] = 2$ as there are two paths inside `gee()`. Inside `foo()`, $W[E] = 1$ as it is a return; $W[G] = W[E] + W[F] = 2$, and $W[H] = W[D] \times W[G] = 4$. Similarly, in `main()`, $W[N] = W[I] \times W[K] = 4$, $W[M] = W[D] \times W[L] = 2$, and $W[O] = W[N] + W[M] = 6$, meaning there are 6 whole-program paths. The bottom of Figure 2.8 shows the probability of the red path being taken, which is exactly $\frac{1}{6}$, same for the others. \square

Note that the computed path counts can be directly used in path sampling, even though the weight of node only denotes *the number of paths from the node to the end of its enclosing*

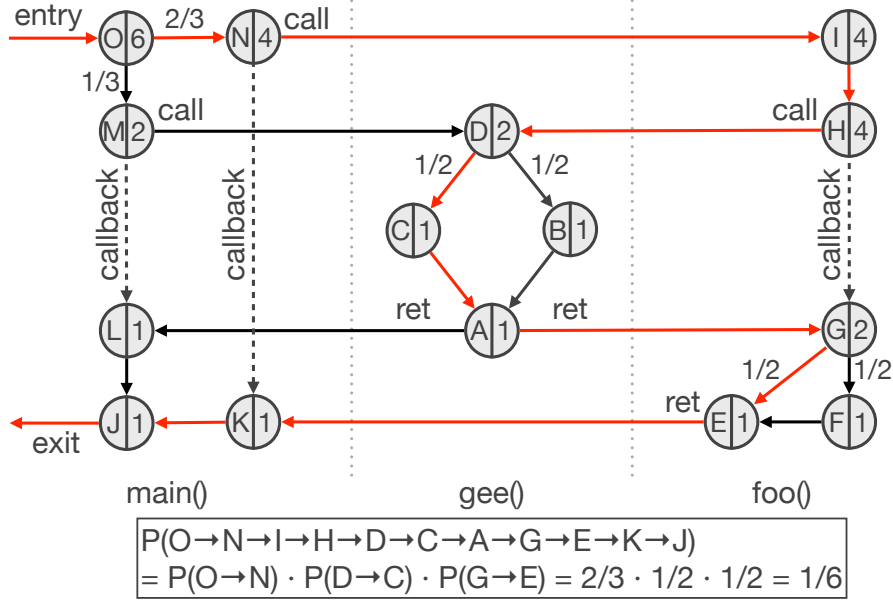


Figure 2.8. Weighted iCFG for Fig. 2.7

function (denoted as x), not *the number of paths from the node to the end of the program* (denoted as y). The reason is that y equals to x times *the number of continuation paths of the enclosing function* (denoted as z), multiplying the same z on both branches of a predicate does not change sample bias. Consider the example in Figure 2.8, nodes C and B have weight 1 (i.e., $x = 1$) although there are 2 paths from either to the end of the program (i.e., $y = 2$). However, using either scheme yields the sampling bias at D (i.e., 1 against 1 versus 2 against 2).

2.4.2 Path Sampling and Probability Analysis

Given the pre-computed weights, our path sampling is to toss a biased coin at a predicate. The predicate bias is locally computed from the weights of the predicate and its children. Since there are substantial variations in weight values (e.g., 1 versus 2^{1000}), we have to design a special procedure to simulate the biased distribution, which is presented in Algorithm 2. In the subsequent section, we will show how to achieve uniform distribution for whole-program path sampling using this algorithm and demonstrate its correctness and

effectiveness. To simplify discussion, we only consider sampling a predicate of two branches, whose weights are w_0 and w_1 with $w_0 > w_1$ without losing generality. The algorithm is to simulate picking branch 0 with the (approximate) probability of $\frac{w_0}{w_0+w_1}$ and branch 1 with $\frac{w_1}{w_0+w_1}$. Sampling more branches can be easily extended. Due to the frequent invocation of the sampling function (for each predicate), we develop an efficient algorithm with $O(1)$ expected complexity (not worst-time complexity). Observe that what we need is a ratio between weights, instead of precise weights. Inspired by the floating point representation, we introduce an approximate representation of weights. Specifically, each weight is transformed to two 64-bit values: *sig* and *exp*, analogous to the significant and exponent in floating point representation, respectively. They satisfy the following, with \widetilde{w}_v an approximation of weight value w_v .

$$\langle \text{sig}, \text{exp} \rangle = \text{sig} \times 2^{\text{exp}} = \widetilde{w}_v \quad (2.2)$$

To minimize representation error, *sig* and *exp* are derived as follows.

$$\begin{cases} \text{exp} = \max(\lfloor \log w_v \rfloor, 63) - 63 \\ \text{sig} = \lfloor w_v / 2^{\text{exp}} \rfloor \end{cases} \quad (2.3)$$

Taking $2^{65} - 1$ as an example, it is represented as $\langle 2^{64} - 1, 1 \rangle$, which introduces an error of $\frac{(2^{64}-1) \times 2^1}{2^{64}-1} = 2.7e - 20$. With the representation, Algorithm 2 describes the sampling procedure. Specifically, if the exponent difference between w_0 and w_1 is smaller than 64, in line 12, BDA randomly samples a value in $[0, \widetilde{w}_0.\text{sig} \times 2^n + \widetilde{w}_1.\text{sig}]$ and then checks if it is smaller than $\widetilde{w}_1.\text{sig}$. If so, branch 1 is selected; otherwise 0, denoting the probability of $\frac{\widetilde{w}_1.\text{sig}}{\widetilde{w}_0.\text{sig} \times 2^n + \widetilde{w}_1.\text{sig}}$. When the exponent difference is larger than 64, it first leverages a loop in lines 5-9 that tosses a fair coin n times and selects 0 when any of the n coins is 0. If all n tries yield 1, which has the probability of $\frac{1}{2^n}$, line 10 further samples with a probability of $\frac{\widetilde{w}_1.\text{sig}}{\widetilde{w}_0.\text{sig}}$, to approximate the intended probability, as w_1 is very small compared to w_0 and hence it is negligible when added to w_0 .

Algorithm 2 Branch Selection

INPUT:	w_0, w_1 :	▷ weights with $w_0 \geq w_1$ without losing generality
OUTPUT:	0/1:	▷ the branch to choose
LOCAL:	\tilde{w}_i : $\langle \text{sig}, \text{exp} \rangle$	▷ approximate representation of weight, consisting of significant bit and exponent

```

1: function SELECTBRANCH( $w_0, w_1$ )                                ▷ Random pick one ID based on weight
2:    $(\tilde{w}_0, \tilde{w}_1) \leftarrow (\text{approximate}(w_0), \text{approximate}(w_1))$ 
3:    $n \leftarrow \tilde{w}_0.\text{exp} - \tilde{w}_1.\text{exp}$ 
4:   if  $n \geq 64$  then
5:     for  $i$  in range( $n$ ) do
6:       if random(2) = 0 then                                ▷ random( $n$ ) returns  $k$  ( $0 \leq k < n$ ) with probability  $\frac{1}{n}$ 
7:         return 0
8:       end if
9:     end for
10:    return (random( $\tilde{w}_0.\text{sig}$ ) <  $\tilde{w}_1.\text{sig}$ )                                ▷  $\tilde{w}_0.\text{sig} \times 2$  must be larger than  $\tilde{w}_1.\text{sig}$ 
11:  else
12:    return (random( $\tilde{w}_0.\text{sig} \times 2^n + \tilde{w}_1.\text{sig}$ ) <  $\tilde{w}_1.\text{sig}$ )
13:  end if
14: end function

```

Theorem 2.4.1. *Using Algorithm 2, the probability \tilde{p} of any whole-program path being sampled satisfies equation 2.4, in which n is the total number of whole-program paths and L is the length of the longest path.*

$$\left(\frac{2^{63}}{2^{63}+1}\right)^{2L} \cdot \frac{1}{n} \leq \tilde{p} \leq \left(\frac{2^{63}+1}{2^{63}}\right)^{2L} \cdot \frac{1}{n} \quad (2.4)$$

Proof. First, for any weight w_v , we prove that \tilde{w}_v follows $\frac{2^{63}}{2^{63}+1} \cdot w_v \leq \tilde{w}_v \leq w_v$.

According to equation 2.3, if $w_v < 2^{64}$, $\tilde{w}_v = w_v$. Otherwise, $\text{sig} \leq w_v/2^{\text{exp}} < \text{sig} + 1$, and hence $\text{sig} \times 2^{\text{exp}} \leq w_v < (\text{sig} + 1) \times 2^{\text{exp}}$. As $\text{sig} \geq 2^{63}$ when $w_v \geq 2^{64}$, we have $\tilde{w}_v \leq w_v < \frac{2^{63}+1}{2^{63}} \cdot \tilde{w}_v$. Thus, $\frac{2^{63}}{2^{63}+1} \cdot w_v \leq \tilde{w}_v \leq w_v$. As a result, the following holds.

$$\frac{2^{63}}{2^{63}+1} \cdot \frac{w_1}{w_1 + w_0} \leq \frac{\tilde{w}_1}{\tilde{w}_1 + \tilde{w}_0} \leq \frac{2^{63}+1}{2^{63}} \cdot \frac{w_1}{w_1 + w_0} \quad (2.5)$$

Let $p_1 = \frac{w_1}{w_1+w_0}$ be the accurate probability of choosing branch 1, the lighter-weight branch. $p_0 = \frac{w_0}{w_1+w_0}$ choosing the other. Thus, we can derive the following 2.6 from inequality 2.5.

$$\frac{2^{63}}{2^{63}+1} \cdot p_l \leq \frac{\widetilde{w}_1}{\widetilde{w}_1 + \widetilde{w}_0} \leq \frac{2^{63}+1}{2^{63}} \cdot p_l \quad (2.6)$$

Next, we derive the bounds of \widetilde{p}_1 , the probability of Algorithm 2 choosing branch 1. There are two cases.

(a) If $n < 64$, we directly have $\widetilde{p}_l = \widetilde{w}_1 / (\widetilde{w}_1 + \widetilde{w}_0)$. According to inequality 2.6, we have the following.

$$\frac{2^{63}}{2^{63}+1} \cdot p_l \leq \widetilde{p}_l \leq \frac{2^{63}+1}{2^{63}} \cdot p_l \quad (2.7)$$

(b) If $n \geq 64$, $\widetilde{p}_1 = \frac{\widetilde{w}_1 \cdot \text{sig}}{\widetilde{w}_0 \cdot \text{sig} \times 2^n}$. Note that $\frac{\widetilde{w}_1}{w_0+w_1} = \frac{\widetilde{w}_1 \cdot \text{sig}}{w_0 \cdot \text{sig} \times 2^n + w_1 \cdot \text{sig}}$. Thus, we have $\widetilde{p}_1 \geq \frac{\widetilde{w}_1}{(w_1+w_0)}$. Combining with inequality 2.6, we can have $\widetilde{p}_1 \geq \frac{2^{63}}{2^{63}+1} \cdot p_l$. On the other hand, $\widetilde{p}_1 = \frac{\widetilde{w}_1}{w_0+w_1} \cdot \frac{\widetilde{w}_0 \cdot \text{sig} \times 2^n + \widetilde{w}_1 \cdot \text{sig}}{w_0 \cdot \text{sig} \times 2^n}$. Because $\widetilde{w}_1 \cdot \text{sig} < 2^{64} \leq 2 \cdot \widetilde{w}_0 \cdot \text{sig}$, we can have $\frac{\widetilde{w}_0 \cdot \text{sig} \times 2^n + \widetilde{w}_1 \cdot \text{sig}}{w_0 \cdot \text{sig} \times 2^n} < \frac{\widetilde{w}_0 \cdot \text{sig} \times 2^n + \widetilde{w}_0 \cdot \text{sig} \times 2}{w_0 \cdot \text{sig} \times 2^n} = \frac{2^{n-1}+1}{2^{n-1}}$. As $n \geq 64$ here, we can have $\widetilde{p}_1 = \frac{\widetilde{w}_1}{w_0+w_1} \cdot \frac{\widetilde{w}_0 \cdot \text{sig} \times 2^n + \widetilde{w}_1 \cdot \text{sig}}{w_0 \cdot \text{sig} \times 2^n} < \frac{\widetilde{w}_1}{w_0+w_1} \cdot \frac{2^{63}+1}{2^{63}}$. Combining with inequality 2.6, we can have $\widetilde{p}_1 < (\frac{2^{63}+1}{2^{63}})^2 \cdot p_l$. Thus,

$$\frac{2^{63}}{2^{63}+1} \cdot p_1 \leq \widetilde{p}_1 \leq (\frac{2^{63}+1}{2^{63}})^2 \cdot p_1 \quad (2.8)$$

From inequality 2.7 and 2.8, the following is true.

$$(\frac{2^{63}}{2^{63}+1})^2 \cdot p_1 \leq \widetilde{p}_1 \leq (\frac{2^{63}+1}{2^{63}})^2 \cdot p_1 \quad (2.9)$$

Similarly, we can prove the bound for \widetilde{p}_0 .

Note that any sampled path could contain at most L conditional predicates. Thus, the probability \widetilde{p} of any whole-program path being sampled satisfies equation 2.4.

□

By applying TAYLOR'S THEOREM to inequality (2.4), we can derive inequality (2.10). In practice, the length L of the longest path of any binary executable (without loops or

recursion) satisfies $L \ll (2^{63} + 1)$, the approximation is hence very tight around $\frac{1}{n}$. For example, 176.gcc’s longest path is nearly 40000, such that $\frac{1-(8e-15)}{n} \leq \tilde{p} \leq \frac{1+(8e-15)}{n}$.

$$\left(1 - \frac{2L}{2^{63} + 1}\right) \cdot \frac{1}{n} \leq \tilde{p} \leq \left(1 + \frac{2L}{2^{63} + 1}\right) \cdot \frac{1}{n} \quad (2.10)$$

We should note that a simple random number generator would not work because of the limitation of floating point representation. Considering selecting a branch with possibility 1e−1000 represented via 1 : 1e+1000, it would be transformed to 2e−308 (the minimal representable positive value in float64), suggesting over 2e+692 times undesirable amplification of the likelihood. This would lead to heavily biased sampling. [27] proposed a heavy-weight algorithm to accurately sample from strongly-biased distribution, whose average-case time complexity is $O(\log(p + q))$ when sampling from $p : q$. In contrast, Algorithm 2 samples in $O(1)$ with negligible precision loss, and hence is more desirable in our context where the sampling function is frequently invoked.

Probabilistic Guarantee for Disclosing Dependence. As mentioned in Section 2.2, a (memory) dependence may be disclosed by many paths. Assume m out of total n paths disclose a dependence, and let $k = \frac{m}{n}$. Following our path sampling algorithm, in a path sample, the probability p_d of observing a given dependency d satisfies inequality (2.11).

$$\left(\frac{2^{63}}{2^{63} + 1}\right)^{2L} \cdot k \leq p_d = \tilde{p} \cdot m \leq \left(\frac{2^{63} + 1}{2^{63}}\right)^{2L} \cdot k \quad (2.11)$$

For N samples, the probability P_d of disclosing dependency d at least once has a lower bound mentioned in inequality (2.12).

$$P_d = 1 - (1 - p_d)^N \geq 1 - \left(1 - \left(\frac{2^{63}}{2^{63} + 1}\right)^{2L} \cdot k\right)^N \approx 1 - (1 - k)^N \quad (2.12)$$

Inequality (2.12) offers a strong guarantee for finding dependency in practice. Taking 176.gcc as an example, if $L=40000$, $k=0.0005$ and $N=10000$, we would have $P_d \geq 99.32\%$, which means that the chance of missing the dependence is only 0.68%.

2.4.3 Addressing Practical Challenges

Handling Loops. Our discussion so far assumes loop-free and recursion-free programs. BDA distinguishes two kinds of loops and handles them differently. The first kind is loops whose iteration numbers are not external input related. We call it *constant loops*. The other kind is input related, called *input-dependent loops*.

For an input-dependent loop, it is intractable to determine how many times it iterates. A standard solution is to compute a fix-point, which often entails substantial over-approximation. Hence, our design is to bound the number of iterations. A naive solution is to give a fixed bound for all input-dependent loops. However, this could cause non-trivial path explosion in the presence of nesting loops. Hence, we bound the total number of iterations across all the nesting loops within a function. Such a design also allows easy computation of weight values. Assume the bound for each function is $t = 3$, Figure 2.9a illustrates the idea. For each function F , BDA clones the function t times, denotes as F_0, \dots, F_{t-1} . For each back-edge in F_i , we reconnect it to the corresponding loop head in F_{i+1} . For example, back-edge a in Figure 2.9 becomes a_1, a_2 and a_3 connecting different versions of F . Note that in the transformed graph at most $t = 3$ back-edges could be taken (e.g., a 3 times and b 0 times; a 2 times and b 1 time; and so on).

For constant loops, which are commonly used in initialization, BDA allows them to iterate as many times as they are supposed to. As such, the constant loop predicates are not part of the path samples generated in this phase. We will show in the next section that our abstract interpreter directly handles such loops without referring to path sample. Recursion is handled in a way similar to input-dependent loops. Details are elided.

Handling Multi-Exits. So far we assume every function in *iCFG* returns to its caller. In practice, many functions may just exit without returning, posing challenges for path counting. Our solution is to count the paths that must exit without return and those that must return separately. We use the sample graph on the top of Figure 2.9b to illustrate the basic idea. In the graph, `main()` (on the left) calls `foo()` (on the right), which may exit without return. The essence of our solution is to count the two sub-graphs below separately and sum them up. Specifically, the sub-graph in the middle corresponds to the must-return

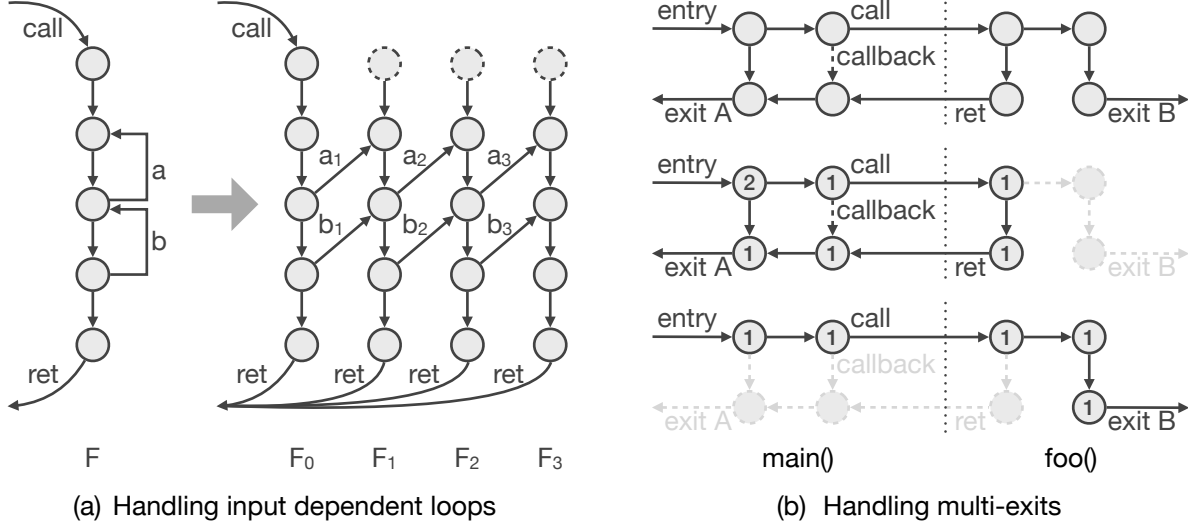


Figure 2.9. Example to show how graph transformation works for loop

behavior, whereas the sub-graph on the bottom corresponds to the exit behavior. The number inside each node denotes its weight. As such, there are $2 + 1 = 3$ whole-program paths.

Handling Indirect Calls. We use an iterative method to handle indirect calls. Specifically, after initial path sampling, BDA abstract-interprets the samples. If new call targets are identified during abstract interpretation, the *iCFG* is updated, weights are re-computed, and another round of sampling is performed. The sampling algorithm terminates when no new indirect-calls are found within a time budget. For example, 7836 indirect calls are identified for 254.gap (by BDA) within 5.67 hours, as shown Table 2.9.

Edge Coverage. If we strictly follow the unbiased whole-program path sampling algorithm, some statements may not be covered. Consider a predicate with two branches, one has weight 1 and the other has weight 2^{1000} . The statement in the short branch may not be covered at all. To address the problem, BDA collects a set of additional samples (usually much smaller than the path samples) to cover control flow edges that have not been covered.

Sampling External Inputs. External input values are sampled from pre-defined value ranges in a uniform fashion. One can consider this input sampling procedure produces an

external input valuation that assigns each instance of an external input read instruction with a random value.

Probabilistic Guarantees in Practice. With the additional machinery to handle practical challenges, Theorem 2.4.1 and bound (2.12) only hold when the following assumption is satisfied: *the graph transformations to handle loops and recursion must not change k , the probability a dependence is disclosed by a whole-program path*. Furthermore, k may be difficult to derive in practice due to the undecidable nature of the problem. However, our experiment in Section 2.7 illustrates that the algorithm is effective in practice and the results are consistent with our theoretical analysis.

2.5 Abstract Interpretation

We explain the abstract interpretation semantics in this section. Given a predefined sample path, represented by a sequence of addresses, and an external input valuation that associates each instruction instance that reads external input with a value sampled from some pre-defined distributions, the abstract interpreter follows the path to compute abstract values for each instruction instance. It models both register and memory reads and writes, e.g., supporting writing an abstract value to an abstract address. If the branch outcome of a loop predicate is not dependent on any external input (e.g., loop predicate with a constant loop bound), BDA does not resort to the path sample, but rather follows the branch based on the abstract value of the predicate. It explicitly represents and updates an abstract call stack, in order to precisely represent stack memory addresses. In addition, the interpretation of arithmetic operations (e.g., additions and subtractions) is precise, without causing any precision loss as that in computing strided intervals in VSA.

Abstract interpretation is essential for BDA. In contrast, an alternative design of using concrete execution to expose dependence is less desirable. Note that in concrete execution, without knowing input specification, the sampled inputs may not satisfy format constraints, leading to early termination. Additionally, concrete execution may have stack/heap reuse, leading to substantial false dependences in the whole-program posterior-analysis, which is necessary and will be explained in Section 2.6.


```

 $\langle \text{Program} \rangle P ::= S$ 

 $\langle \text{Statement} \rangle S ::= S_1; S_2 \mid r := e \mid r := \mathbf{R}(r_a) \mid \mathbf{W}(r_a, r_v) \mid r := \mathbf{malloc}() \mid r :=$ 
 $\quad \mathbf{input}() \mid \mathbf{call}(a) \mid \mathbf{ret} \mid \mathbf{goto}(a) \mid \mathbf{if } r \mathbf{ then goto}(a)$ 

 $\langle \text{Expression} \rangle e ::= r \mid v \mid r_1 \text{ op } r_2 \mid r \text{ op } v$ 

 $\langle \text{Operator} \rangle \text{op} ::= + \mid - \mid * \mid / \mid \dots$ 

 $\langle \text{Register} \rangle r ::= \{sp, r_1, r_2, \dots\}$ 

 $\langle \text{AbstractValue} \rangle v ::= \langle m, c \rangle$ 

 $\langle \text{MemoryRegion} \rangle m ::= \mathcal{G} \mid \mathcal{H}_a^c \mid \mathcal{S}_a^c$ 

 $\langle \text{Const} \rangle c ::= \{0, 1, 2, \dots\}$ 

 $\langle \text{Address} \rangle a ::= \{0, 1, 2, \dots\}$ 

```

Figure 2.10. Language

Language. To facilitate discussion, we introduce a low-level language to model binary executables. The language is designed to illustrate our key ideas, and hence omits many features (of x86). The implementation of BDA supports these complex features present in real-world binary executables (even though they may not be modeled by our language). The syntax of the language is shown in Figure 2.10. $\mathbf{R}(r_a)$ and $\mathbf{W}(r_a, r_v)$ model memory read and write operations, respectively, where register r_a holds the address and register r_v holds the value to write. Heap allocation functions (e.g., `calloc` and `mmap`) are modeled as `malloc`. The allocated size is irrelevant in our analysis and hence elided. External input functions (e.g., `fread` and `scanf`) are modeled by `input`. Other general function calls and returns are modeled by `call` and `ret`. The address of the target function of `call` is a . We assume parameter passing across functions is done explicitly through register and memory read/write instructions. We model the stack pointer register sp to facilitate computing stack related abstract values. In addition, control flow statements (in high-level languages), such as conditional and loop statements, are modeled using `goto` and guarded `goto`.

Abstract values are represented as $\langle m, c \rangle$, where m stands for a memory region and c stands for the offset relative to the base of the region. The memory space is partitioned to three disjoint regions: global, stack and heap. The global region, denoted as \mathcal{G} , stands for the locations holding initialized and uninitialized global data, such as the `.data`, `.rodata` and `.bss` segments of an ELF file. A stack region, denoted as \mathcal{S}_a^c , models a stack frame that holds local variable values for the c -th invocation instance of the function at address a . A heap region, denoted as \mathcal{H}_a^c , models a memory region allocated in the c -th invocation instance of the allocation instruction at *program counter* (pc) address a . A non-address constant value can be expressed as having $m = \mathcal{G}$. Note that in our interpretation, an instruction may be encountered multiple times in a sample path and we distinguish these different instances. In contrast, VSA does not; instead it merges the abstract values for all possible instances, which is an important source of inaccuracy.

Definitions. Figure 2.11 introduces a number of definitions that are used in the semantic rules. We use pc to denote the program counter that indicates the address of current instruction, IS to denote the size of each instruction, IC to represent the current instance of an instruction, and LP to indicate whether the current instruction is a loop predicate. MS denotes the abstract value store that maps an abstract memory address value to the abstract value stored at that address, and RS denotes the register store that maps a register to its abstract value. MT and RT represent the taint stores for memory and registers, respectively. The taint tag of an abstract value indicates if the value has been directly/-transitively computed from some (randomly sampled) external input. In other words, there is data flow from some external inputs to the abstract value. A sample path is denoted by PA , which is a list of addresses ordered by their appearance in the path. A sampled external input valuation RV assigns a sampled value to each instance of an external input instruction. Both PA and RV are generated by the previous sampling phase and provided as inputs to the abstract interpretation process. We use CS to explicitly model call stack. It is a list of four-element tuples, denoting respectively the invocation site, its instance, the return address, and a copy of the abstract value of the sp register which is supposed to be

```

 $pc \in \text{ProgramCounter} ::= \text{Address}$ 
 $IS \in \text{InstructionSize} ::= \text{Address} \rightarrow \text{Const}$ 
 $IC \in \text{InvocationCount} ::= \text{Address} \rightarrow \text{Const}$ 
 $LP \in \text{LoopPredicate} ::= \text{Address} \rightarrow \text{Bool}$ 
 $MS \in \text{MemStore} ::= \text{AbstractValue} \rightarrow \text{AbstractValue}$ 
 $RS \in \text{RegStore} ::= \text{Register} \rightarrow \text{AbstractValue}$ 
 $MT \in \text{MemTaint} ::= \text{AbstractValue} \rightarrow \text{Bool}$ 
 $RT \in \text{RegTaint} ::= \text{Register} \rightarrow \text{Bool}$ 
 $PA \in \text{PATH} ::= [\text{Address}]$ 
 $RV \in \text{RandomInputValuation} ::= (\text{Address} \times \text{Const}) \rightarrow \text{Const}$ 
 $CS \in \text{CallStack} ::= [\text{Address} \times \text{Integer} \times \text{Address} \times \text{AbstractValue}]$ 
 $MOS \in \text{MemOpSeq} ::= [\text{Address} \times \text{AbstractValue}]$ 

```

<pre> CalcValue(op, v_1, v_2) ::= if $v_1.m \equiv \mathcal{G}$ then $v_3 \leftarrow \langle v_2.m, v_1.c \text{ op } v_2.c \rangle;$ $t \leftarrow \text{false};$ else if $v_2.m \equiv \mathcal{G}$ then $v_3 \leftarrow \langle v_1.m, v_1.c \text{ op } v_2.c \rangle;$ $t \leftarrow \text{false};$ else $v_3 \leftarrow \langle \mathcal{G}, RV[\langle pc, IC[pc] \rangle] \rangle;$ $t \leftarrow \text{true};$ end if return $\langle v_3, t \rangle;$ </pre>	<pre> NormalizeVal(v) ::= if $v.m \equiv S_*^*$ then $CS' \leftarrow CS;$ while $v.c \geq 0$ and $\neg CS'.\text{empty}()$ do $\langle -, -, -, v_t \rangle \leftarrow CS'.\text{pop}();$ $v.m \leftarrow v_t.m;$ $v.c \leftarrow v.c + v_t.c;$ end while end if return $v;$ </pre>
--	---

Figure 2.11. Definitions

updated upon function invocation. The outcome of abstract interpretation MOS contains the abstract values for each memory access instruction encountered.

Semantics Rules. The semantic rules are presented in Table 2.2. Upon interpreting an instruction, the instance count IC is incremented by one. Rule **Read** describes the semantics of memory read. It invokes an auxiliary procedure **NormalizeVal()** to normalize the abstract (address) value in register r_a , denoted as $RS[r_a]$. As shown in Figure 2.11, if the value is a global or heap value, it is returned directly. Otherwise, it is checked to identify the enclosing stack frame of the address. Note that it is common for an instruction to access

a stack location beyond the current stack frame (e.g., access an argument passed from the caller function). The procedure traverses the stack frames from the top to the bottom till it finds a frame on which the offset becomes negative. After normalization, the abstract value stored in the normalized address is copied to the target register r . The taint bit of r is the union of the taint bits of the normalized address and the address register r_a . At the end, the pc is updated to the next instruction. Rule **Write** describes the semantics of memory write. Similar to memory read, it normalizes the address value and then updates the memory value store MS and the memory taint store MT . Rule **Malloc** creates a new abstract value denoting the allocation site with 0 offset. Note that BDA does not model memory safety and hence the size of allocation is irrelevant. Intuitively, one can consider each allocated heap region has infinite size. This can be achieved during abstract interpretation but not during concrete execution. Rule **Input** loads the abstract value of destination register r from the pre-generated external input sample valuation RV , which is constructed by drawing value samples from predefined distributions during the preceding sampling phase. In addition, the taint bit is set **true** to indicate that the value is related to external input. Rule **Goto** sets the program counter to the target address a .

In Rule **If-Goto**, if the taint bit of r is not set and the current instruction is a loop predicate, that is, r is not directly/transitively computed from external input, the loop branch outcome is certain and independent from the sampled value. Hence, pc is set to a_t , which is either the branch target a specified by the statement when r is true, or the fall-through address. Otherwise, it is loaded from the pre-computed path sample PA . Observe that BDA respects path feasibility when loop predicate outcome is not derived from any external input, e.g., *constant loops* (in the initialization phase). Taint analysis allows us to identify such predicates. In Rule **Call**, pc is first copied to pc' , then it is updated by loading from the sample path PA . BDA may determine to skip a function call if it is part of a recursion. If the call is not skipped, indicated by pc being equal to the specified target a , the invocation site pc' , its instance count, the return address (i.e., the instruction after the invocation), and the current abstract value of sp are pushed to the call stack CS . Then, the abstract value of sp is reset, indicating a new stack frame. Rule **Ret** pops the call stack to acquire the return address and restores the value of sp . Rules **Expr1** and **Expr2** update

Table 2.2. Interpretation rules

Rule	Statement	Actions
Read	$r := \mathbf{R}(r_a)$	$IC[pc]++$; $v := \mathbf{NormalizeVal}(RS[r_a])$; $RS[r] := MS[v]$; $RT[r] := MT[v] \vee RT[r_a]$; $MOS.enqueue(\langle pc, v \rangle)$; $pc := pc + IS[pc]$;
Write	$\mathbf{W}(r_a, r_v)$	$IC[pc]++$; $v := \mathbf{NormalizeVal}(RS[r_a])$; $MS[v] := RS[r_v]$; $MT[v] := RT[r_v] \vee RT[r_a]$; $MOS.enqueue(\langle pc, v \rangle)$; $pc := pc + IS[pc]$;
Malloc	$r := \mathbf{malloc}()$	$IC[pc]++$; $RS[r] := \langle H_{pc}^{IC[pc]}, 0x0 \rangle$; $pc := pc + IS[pc]$;
Input	$r := \mathbf{input}()$	$IC[pc]++$; $RS[r] := RV[pc, IC[pc]]$; $RT[r] := \mathbf{true}$; $pc := pc + IS[pc]$;
Goto	$\mathbf{goto}(a)$	$IC[pc]++$; $pc := a$;
If-Goto	$\mathbf{if } r \mathbf{ then goto}(a)$	$IC[pc]++$; $a_t := (RS[r] \neq \langle \mathcal{G}, 0 \rangle ? a : pc + IS[pc])$; $pc := \neg RT[r] \wedge LP(pc) ? a_t : PA.pop()$;
Call	$\mathbf{call}(a)$	$IC[pc]++$; $pc' := pc$; $pc := PA.pop()$; $\mathbf{if } (pc == a) \{$ $CS.push(pc', IC[pc'], pc' + IS[pc'], RS[sp])$; $RS[sp] := \langle S_{pc}^{IC[pc]}, 0x0 \rangle$; $\}$;
Ret	\mathbf{ret}	$IC[pc]++$; $\langle -, -, pc, RS[sp] \rangle := CS.pop()$;
Expr1	$r_t := r_1 \text{ op } r_2$	$IC[pc]++$; $\langle RS[r_t], t \rangle := \mathbf{CalcValue}(op, RS[r_1], RS[r_2])$; $RT[r_t] := RT[r_1] \vee RT[r_2] \vee t$; $pc := pc + IS[pc]$;
Expr2	$r_t := r \text{ op } v$	$IC[pc]++$; $\langle RS[r_t], t \rangle := \mathbf{CalcValue}(op, RS[r], v)$; $RT[r_t] := RT[r] \vee t$; $pc := pc + IS[pc]$;

the resulting register r_t with the value calculated by the **CalcValue()** procedure and record the corresponding taint tags. As shown in Figure 2.11, **CalcValue()** computes the result of operation op on operands v_1 and v_2 . If one of the operands belongs to the global region, then the resulting memory region is inherited from the other operand and the resulting offset is derived by performing the operation on the offset fields of the two operands. Otherwise (e.g., both operands denote values in some heap region, which may occur as path feasibility may not be respected by BDA), we use a random value as the result, since we could not obtain a precise result for operations on two non-global abstract values. In this case, the result taint tag is set to **true**.

Example. Consider the example in Table 2.3. The source code, the source level trace, the trace in our language, and the interpretation actions are shown in the columns from left to

Table 2.3. Abstract interpretation example ($PA = a \rightarrow e \rightarrow g \rightarrow k$)

SourceCode	Trace	BDA Trace	Actions
<pre> 1 . int main() { 2 . char *s = malloc(2); 3 . foo(s); 4 . } 5 . 6 . void foo(char *s) { 7 . if(input()) return; 8 . gee(s); 9 . } 10. 11. void gee(char *s) { 12. for(int i=0; i<2; i++) 13. s[i] = input(); 14. }</pre>	2	a. $r_1 := \text{malloc}()$	$RS \quad [r_1] = \langle \mathcal{H}_a^1, 0 \rangle$
		b. $sp := sp - \langle \mathcal{G}, 4 \rangle$	$RS \quad [sp] = \langle \mathcal{S}_a^1, -4 \rangle$
		c. $\mathbf{W}(sp, r_1)$	$MS \quad [\langle \mathcal{S}_a^1, -4 \rangle] = \langle \mathcal{H}_a^1, 0 \rangle$
	3	d. $\text{call}(e)$	$RS \quad [sp] = \langle \mathcal{S}_e^1, 0 \rangle$
			$CS \quad [\langle \mathcal{S}_a^1, -4 \rangle]$
	7	e. $r_3 := \text{input}()$	$RS \quad [r_3] = \langle \mathcal{G}, 502 \rangle$
		f. $\text{if } r_3 \text{ then goto}(p)$	
	8	g. $r_2 := \mathbf{R}(sp)$	$RS \quad [r_2] = \langle \mathcal{H}_a^1, 0 \rangle$
		h. $sp := sp - \langle \mathcal{G}, 4 \rangle$	$RS \quad [sp] = \langle \mathcal{S}_e^1, -4 \rangle$
		i. $\mathbf{W}(sp, r_2)$	$MS \quad [\langle \mathcal{S}_e^1, -4 \rangle] = \langle \mathcal{H}_a^1, 0 \rangle$
		j. $\text{call}(k)$	$RS \quad [sp] = \langle \mathcal{S}_k^1, 0 \rangle;$ $CS \quad [\langle \mathcal{S}_a^1, -4 \rangle, \langle \mathcal{S}_e^1, -4 \rangle]$
	11	k. $r_4 := \mathbf{R}(sp)$	$RS \quad [r_4] = \langle \mathcal{H}_a^1, 0 \rangle$
	12	l. $r_5 := \langle \mathcal{G}, 0 \rangle$	$RS \quad [r_5] = \langle \mathcal{G}, 0 \rangle$
		m. $r_6 := r_5 \geq \langle \mathcal{G}, 2 \rangle$	$RS \quad [r_6] = \langle \mathcal{G}, 0 \rangle$
		n. $\text{if } r_6 \text{ then goto}(x)$	

right. Observe that instructions $a - c$ correspond to the invocation at line 2 that writes the returned value from `malloc()` to stack. In d (i.e., invocation to `foo()` in line 3), the current sp value $\langle \mathcal{S}_a^1, -4 \rangle$ is pushed to CS ; sp is updated to denote the stack frame of `foo()`; and the target instruction e is loaded from the sample path PA (in the caption of Table 2.3). In f (i.e., the conditional in line 7), since r_3 is from external input, the target g is loaded from PA . In g (i.e., passing s in line 8), when sp (i.e., the address of local variables) is read, its value $\langle \mathcal{S}_e^1, 0 \rangle$ is first normalized to $\langle \mathcal{S}_a^1, -4 \rangle$, which is used to access MS to acquire the value of $\langle \mathcal{H}_a^1, 0 \rangle$. In n (i.e., the loop predicate in line 12), r_6 is not input related, the interpreter evaluates the predicate and takes the true branch. \square

2.6 Posterior Analysis

After the abstract interpretation of all sampled paths, the posterior analysis is performed to complete dependence analysis, via aggregating the abstract values collected from individual path samples in a flow-sensitive, context-sensitive, and path-insensitive fashion.

Algorithm 3 Posterior Dependence Analysis

INPUT:	<i>MOSes</i> :	$\{\text{MemOpSeq}\}$	\triangleright set of memory operation sequences
	<i>iCFG</i> :	$\text{Node} \times \text{Edge}$	\triangleright inter-procedural control flow graph
OUTPUT:	<i>DIP</i> :	$\{\text{Address} \times \text{Address}\}$	\triangleright set of dependent instruction pairs
LOCAL:	<i>GI2M</i> :	$\text{Address} \rightarrow \{\text{AbstractValue}\}$	\triangleright map an instruction to abstract addresses accessed by it
	<i>GDEP</i> :	$\text{Address} \rightarrow \{\text{Address}\}$	\triangleright map an instruction to its depending instructions
	<i>GKILL</i> :	$\text{Address} \rightarrow \{\text{Address}\}$	\triangleright map an instruction to reaching definitions killed by it
	<i>M2I</i> :	$\text{AbstractValue} \rightarrow \{\text{Address}\}$	\triangleright map an abstract address to its definitions
	<i>WL</i> :	$[\text{CallString} \times \text{Address}]$	\triangleright work list of program points with calling context
	<i>PS</i> :	$(\text{CallString} \times \text{Address}) \rightarrow (\text{AbstractValue} \rightarrow \{\text{Address}\})$	\triangleright abstract state

```

1: function POSTERIORDEPENDENCEANALYSIS(MOSes, iCFG)
2:   for MOS in MOSes do
3:      $\langle I2M, DEP, KILL \rangle \leftarrow \text{PERSAMPLEANALYSIS}(MOS)$ 
4:      $GI2M \leftarrow \text{map\_merge}(GI2M, I2M)$ 
5:      $GDEP \leftarrow \text{map\_merge}(GDEP, DEP)$ 
6:      $GKILL \leftarrow \text{map\_merge}(GKILL, KILL)$ 
7:   end for
8:    $WL.\text{enqueue}(\langle \text{nil}, \text{entry}(iCFG) \rangle)$ 
9:   while  $\neg WL.\text{empty}()$  do
10:     $\langle cs, iaddr \rangle \leftarrow WL.\text{dequeue}()$ 
11:    if  $\text{is\_call}(iaddr)$  then  $\triangleright$  update calling context upon a call instruction
12:       $cs.\text{push}(iaddr)$ 
13:       $\text{succs} \leftarrow \text{call\_target}(iCFG, iaddr)$ 
14:    else
15:      if  $\text{is\_ret}(iaddr)$  then
16:         $iaddr \leftarrow cs.\text{pop}()$ 
17:      end if
18:       $\text{succs} \leftarrow \text{get\_succ}(iCFG, iaddr)$   $\triangleright$  get the following instruction
19:    end if
20:     $M2I \leftarrow PS[\langle cs, iaddr \rangle]$   $\triangleright$  the set of reaching definitions at  $iaddr$ 
21:    if  $\text{is\_memory\_write}(iaddr)$  then
22:       $M2I \leftarrow \text{HANDLEMEMORYWRITE}(iaddr, M2I, GI2M, GKILL)$ 
23:    else if  $\text{is\_memory\_read}(iaddr)$  then
24:       $DIP \leftarrow \text{HANDLEMEMORYREAD}(iaddr, DIP, M2I, GI2M, GDEP)$ 
25:    end if
26:    for succ in succs do
27:      if  $\neg \text{map\_contains}(PS[\langle cs, succ \rangle], M2I)$  then
28:         $PS[\langle cs, succ \rangle] \leftarrow \text{map\_merge}(PS[\langle cs, succ \rangle], M2I)$ 
29:         $WL.\text{enqueue}(\langle cs, succ \rangle)$   $\triangleright$  additional analysis round is needed when changes detected
30:      end if
31:    end for
32:  end while
33:  return DIP
34: end function

```

Algorithm 4 Handle Memory Write

INPUT:	$iaddr$:	Address	\triangleright the current instruction
	$M2I$:	$\text{AbstractValue} \rightarrow \{\text{Address}\}$	\triangleright map an abstract address to its definitions
	$GI2M$:	$\text{Address} \rightarrow \{\text{AbstractValue}\}$	\triangleright map an instruction to the abstract addresses accessed by it
	$GKILL$:	$\text{Address} \rightarrow \{\text{Address}\}$	\triangleright map an instruction to reaching definitions killed by it
OUTPUT:	$M2I'$:	$\text{AbstractValue} \rightarrow \{\text{Address}\}$	\triangleright a new map between abstract address to definitions

```
1: function HANDLEMEMORYWRITE( $iaddr, M2I, GI2M, GKILL$ )
2:    $M2I' \leftarrow M2I$ 
3:   for  $maddr$  in  $GI2M[iaddr]$  do
4:     if  $\text{capacity}(GI2M[iaddr]) \equiv 1$  then  $\triangleright$  strong update
5:        $M2I'[maddr] \leftarrow \{iaddr\}$ 
6:     else
7:        $M2I'[maddr] \leftarrow M2I'[maddr] \cup \{iaddr\}$ 
8:       if  $\text{capacity}(GKILL[iaddr]) \equiv 1$  then  $\triangleright$  strong kill
9:          $M2I'[maddr] \leftarrow M2I'[maddr] \setminus GKILL[iaddr]$ 
10:      end if
11:    end if
12:  end for
13:  return  $M2I'$ 
14: end function
```

Specifically, it computes *abstract states* for each *program point*, which is an instruction annotated with a calling context. The abstract states represent the set of live abstract addresses at the given program point and their definition instructions (an intuitive correspondence at the source level is that the set of live variables at a program point and the statements that define them). Dependences are detected between a read instruction and all the definitions to the address being read. It is context-sensitive as it considers instructions under different contexts as different program points. It is path-insensitive as it merges the abstract values collected along different branches at control flow joint point (e.g., the instruction where the two branches of a conditional statement meet). This allows addressing incompleteness in path sampling. *However, our analysis is much more accurate than a flow-sensitive and path-insensitive data-flow analysis as it does not compute any new abstract values (e.g., by transfer functions in standard data flow analysis), but rather just aggregates the collected abstract values.* This avoids the substantial precision loss caused by the conservativeness of transfer functions. Intuitively, abstract values collected in individual sample paths are

Algorithm 5 Per-sample Analysis

INPUT:	MOS :	MemOpSeq	▷ memory operation sequence
OUTPUT:	$I2M$:	Address \rightarrow {AbstractValue}	▷ map an instruction to abstract addresses accessed by it
	DEP :	Address \rightarrow {Address}	▷ map an instruction to the instructions it depends on
	$KILL$:	Address \rightarrow {Address}	▷ map an instruction to reaching definitions it kills
LOCAL:	DEF :	AbstractValue \rightarrow Address	▷ map an abstract address to its latest definition

```
1: function PERSAMPLEANALYSIS( $MOS$ )
2:   while  $\neg MOS.empty()$  do
3:      $\langle iaddr, maddr \rangle \leftarrow MOS.dequeue()$   ▷ acquire an instruction instance and the accessed address
4:      $I2M[iaddr] \leftarrow I2M[iaddr] \cup \{maddr\}$ 
5:     if  $is\_memory\_write(iaddr)$  then
6:        $KILL[iaddr] \leftarrow KILL[iaddr] \cup \{DEF[maddr]\}$   ▷ previous definition of  $maddr$  is killed by
        $iaddr$ 
7:        $DEF[maddr] \leftarrow iaddr$   ▷  $iaddr$  is the new definition of  $maddr$ 
8:     else if  $is\_memory\_read(iaddr)$  then
9:        $DEP[iaddr] \leftarrow DEP[iaddr] \cup \{DEF[maddr]\}$   ▷ detect a new dependence
10:    end if
11:  end while
12:  return  $\langle I2M, DEP, KILL \rangle$ 
13: end function
```

propagated through all paths (by the merge operation) to disclose any missing dependences due to incomplete path sampling. To further mitigate the precision loss caused by the merge operation, our analysis also features strong updates [28] and strong kills that preclude bogus abstract states.

Detailed Design. The details of the analysis are shown in Algorithm 3. It takes as input the set of memory operation sequences (MOS es), each sequence generated by interpreting a path sample, and the inter-procedural control flow graph ($iCFG$) that maps an instruction to its successor(s), and produces the instruction pairs with (memory) dependence relations (DIP). The process consists of two stages. In the first stage (lines 2-7), a per-sample analysis (Algorithm 5) is performed on each memory operation sequence to derive three pieces of information: the set of abstract addresses accessed by each instruction $I2M$, the set of definitions (i.e., writes) each instruction depends on DEP , and the set of definitions killed by each write instruction $KILL$. These results are merged to their global correspondences (lines 4-6). In the second stage (lines 8-32), a work list (WL) is used to traverse $iCFG$ to compute abstract states PS for each program point. Lines 11-19 determine the successors of

Algorithm 6 Handle Memory Read

INPUT:	<i>iaddr</i> :	Address	▷ the current instruction
	<i>DIP</i> :	Address × Address	▷ dependencies
	<i>M2I</i> :	AbstractValue → {Address}	▷ map an address to its definitions
	<i>GI2M</i> :	Address → {AbstractValue}	▷ map an instruction to its accessed addresses
	<i>GDEP</i> :	Address → {Address}	▷ map an instruction to its dependences in samples
OUTPUT:	<i>DIP'</i> :	Address × Address	▷ updated dependences

```

1: function HANDLEMEMORYREAD(iaddr, DIP, M2I, GI2M, GDEP)
2:   if capacity(GDEP[iaddr]) ≡ 1 then                                ▷ string dependence
3:     for def in GDEP[iaddr] do
4:       DIP' ← DIP' ∪ {⟨iaddr, def⟩}
5:     end for
6:   else
7:     for maddr in GI2M[iaddr] do
8:       for def in M2I[maddr] do
9:         DIP' ← DIP' ∪ {⟨iaddr, def⟩}
10:      end for
11:    end for
12:  end if
13:  return DIP'
14: end function

```

the current program point and maintain the calling context *cs*. If *iaddr* is a memory write (lines 21-22), the set of live addresses and their definitions *M2I* are updated by the procedure HANDLEMEMORYWRITE() (Algorithm 4). Specifically, Algorithm 4 checks if *iaddr* defines the same abstract address in all sample paths (line 4 in Algorithm 4). If so, strong update is performed by resetting the definition of *maddr* to *iaddr*; otherwise, *iaddr* is added to the definition set of *maddr* (line 7). If *iaddr* always kills the same definition in all samples (line 8), the definition is removed from the result set (line 9). Return to Algorithm 3. In lines 23-24, if *iaddr* is a memory read, dependences are derived from *M2I*, the abstract state at *iaddr*, through the procedure HANDLEMEMORYREAD() (Algorithm 6, which is Similar to Algorithm 4) Lines 26-31 proceed to the succeeding program points. Particularly, a control flow successor is added to the work-list if its abstract state has undertaken any change (lines 27-29). Our analysis terminates when a fixed point is reached. At the end, we want to mention that full context-sensitivity is very expensive. Hence BDA supports configurable call depth. In our experiment, we use depth 2.

```

1  char bar(char *p) {
2      *p = 0;
3      if (input()) {
4          *p = 1;
5          foo(*p);
6      }
7      if (input()) return *p;
8      else return ~(*p);
9  }

```

Figure 2.12. Posterior analysis example

Example. Consider the example in Figure 2.12. For simplicity, we use source code to explain our ideas. Assume BDA collects 2 path samples: $2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 6 \rightarrow 8$ and path $2 \rightarrow 3 \rightarrow 7$. As such, abstract interpretation exposes dependences from line 7 to 4, from line 5 to 4, and from line 8 to 2, but missing that from line 8 to 4 due to incomplete path coverage. By merging the abstract values from the two branches of the predicate in line 3, the posterior analysis discloses the missing dependence. Additionally, as function `bar()` might be invoked several times, pointer `p` at line 4 might have multiple abstract values. As such, at line 4 traditional analysis like VSA cannot kill the definition from line 2, whereas BDA can, by its strong kill. This prevents the bogus dependence from line 5 to 2. \square

2.7 Evaluation

BDA is implemented in Rust, leveraging Radare2 [29] that provides basic disassembling functionalities. For input distribution, we used a fixed normal distribution $\mathcal{N}(\mu = 0, \sigma^2 = 32768^2)$, without assuming prior knowledge¹. To assess BDA’s effectiveness and efficiency, we compare it with Alto and VSA (from state-of-the-art binary analysis platforms) on SPECINT2000, a standard benchmark widely used by binary analysis techniques including the aforementioned two. We also apply BDA in two downstream analyses, one is to identify indirect control flow transfer targets, a critical challenge in constructing call graphs, and the other is to identify hidden malicious behaviors of a set of 12 recent malware samples provided

¹↑We have tried different parameters. The impact is not significant.

Table 2.4. SPECINT2000 programs

Program	Size	# Insn	# Block	# Func
164.gzip	143,760	7,650	707	61
175.vpr	435,888	32,218	2,845	255
176.gcc	4,709,664	378,261	36,931	1,899
181.mcf	62,968	2,977	213	24
186.crafty	517,952	42,084	4,433	104
197.parser	367,384	24,584	2,911	297
252.eon	3,423,984	40,119	7,963	615
253.perlbmk	1,904,632	133,755	12,933	717
254.gap	1,702,848	91,608	9,020	458
255.vortex	1,793,360	109,739	16,970	624
256.bzip2	108,872	6,859	577	63
300.twolf	753,544	57,460	4,280	167

Table 2.5. Malware samples

Malware	Size	Report Date
1a0b96488c4be390ce2072735ffb0e49	1,806,356	2018-03-10
3fb857173602653861b4d0547a49b395	163,099	2018-07-17
49c178976c50cf77db3f6234efce5eeb	116,385	2018-03-12
5e890cb3f6cba8168d078fdede090996	18,112	2018-03-14
6dc1f557eac7093ee9e5807385dbcb05	88,520	2018-07-09
72afccb455faa4bc1e5f16ee67c6f915	729,816	2017-05-17
74124dae8fdbb903bece57d5be31246b	21,804	2018-10-09
912bca5947944fdcd09e9620d7aa8c4a	124,366	2018-10-09
a664df72a34b863fc0a6e04c96866d4c	200,976	2018-07-17
c38d08b904d5e1c7c798e840f1d8f1ee	178,781	2017-02-24
c63cef04d931d8171d0c40b7521855e9	88,436	2018-03-14
dc4db38f6d3c1e751dcf06bea072ba9c	124,154	2018-07-17

by VirtualTotal [30]. In the former experiment, we compare BDA with IDA, an industry standard platform. In the latter, we compare with Cuckoo [23], a state-of-the-art malware analysis platform. All experiments were conducted on a server equipped with 32-cores CPU (Intel® Xeon™ E5-2690 @ 2.90GHz) and 128G main memory. Tables 2.4 and 2.5 present the basic information of the SPECINT2000 binaries and the malware samples.

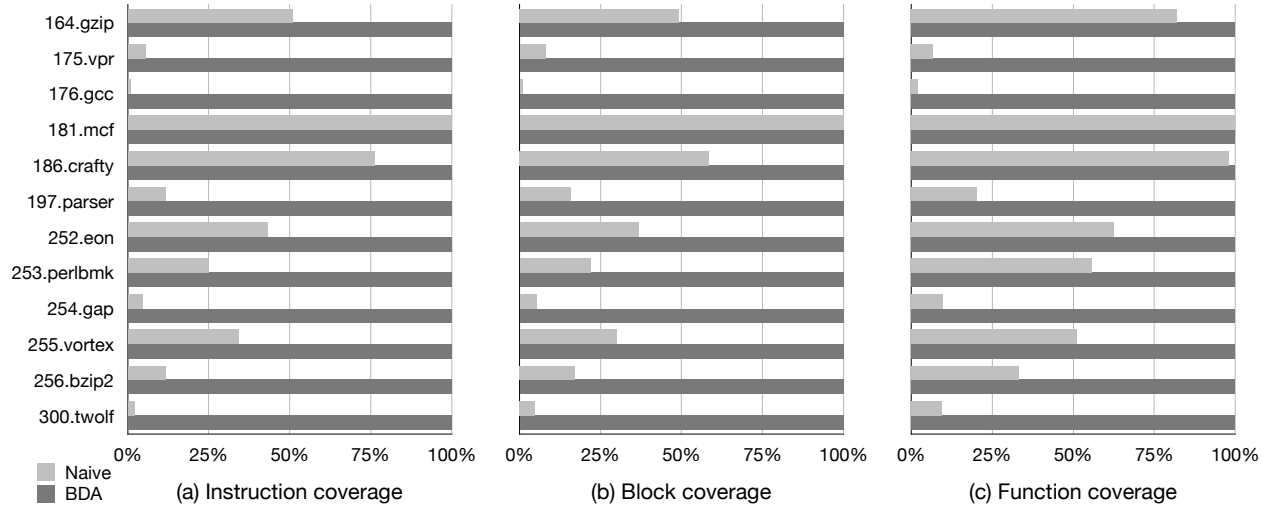


Figure 2.13. Code coverage

2.7.1 Coverage

Code coverage. In this experiment, we study the code coverage of our unbiased whole-program path sampling algorithm and compare it with a naive algorithm that tosses a fair coin at each predicate. Specifically, we collect 10,000 path samples for each algorithm and report the code coverage. Figures 2.13a, 2.13b and 2.13c present the code coverage of our algorithm (in dark gray bars) and the naive algorithm (in the light gray bars) at the instruction level, basic block level and the function level, respectively. Overall, our algorithm can achieve almost 100% coverage for all programs. On average, it covers 554% more instructions, 529% more basic blocks, and 258% more functions than the naive algorithm. For programs with complex path structures, our algorithm has much better coverage. Take 197.parser as an example. It contains lots of error-handling code that detours from the main processing logic at the beginning of the program. The naive algorithm tends to get stuck in these error handling paths. In contrast, our sampling algorithm appropriately prioritizes the main processing logic that contains many more deep paths.

Path coverage. Next, we study the path coverage. Since there are usually an extremely large number of whole-program paths (even not considering loops and recursion), it is not that useful to report whole-program path coverage. Hence, we report intra-procedure path

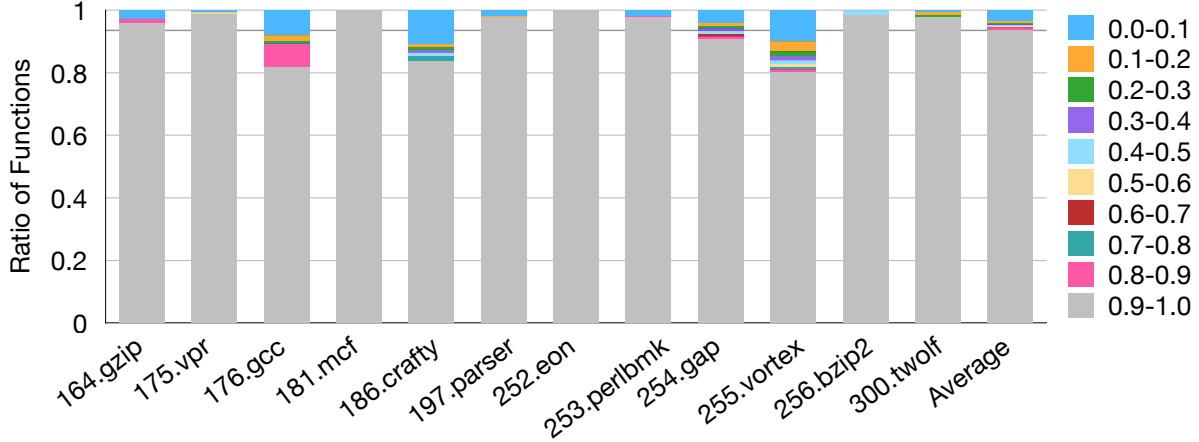


Figure 2.14. Path coverage.

coverage, in which the paths we consider are the BL paths defined in [26]. Specifically, these are intra-procedural paths starting at function entry or some loop heads and ending at function exit or a back-edge. The results are shown in Figure 2.14, which shows the percentage of functions for which BDA has achieved various levels of coverage. As we can see, 93% of the functions have a full or close-to-full path coverage. Those functions whose path coverage is less than 50% have an extremely large number of unique paths (e.g., function `get_method()` in 164.gzip has 4514809836 BL paths). As we will show in the next experiment, according to the observation discussed in Section 2.2 that a dependence tends to be covered by many paths. Incomplete path coverage does not cause prominent problems for us. In addition, the posterior analysis substantially mitigates the issue as well.

2.7.2 Program Dependence

In this experiment, we perform dependence analysis on SPECINT2000 programs. We also compare with Alto and VSA. For Alto, we port its original implementation [31] on DEC Alpha to x86. There are three popular binary analysis platforms that support VSA, including CodeSurfer [16], ANGR [18], and BAP [17]. Among them, CodeSurfer is not publicly available and ANGR’s VSA does not handle complex binaries as SPECINT2000 programs (after confirming with the authors). We hence choose BAP’s VSA for comparison (called

Table 2.6. Memory Dependence

Program	# Refer	Alto				BDA				Reduce
		# Found	# Miss	# Extra	# MisTyped	# Found	# Miss	# Extra	# MisTyped	
164.gzip	3,580	2,229,749	0 (0.00%)	2,226,169	302,100 (13.55%)	29,370	8 (0.22%)	25,798	3,502 (11.92%)	7492%
175.vpr	13,042	36,840,012	0 (0.00%)	36,826,970	26,692,177 (72.45%)	559,460	10 (0.08%)	546,428	346,217 (61.88%)	6485%
181.mcf	2,050	588,076	0 (0.00%)	586,026	324,621 (55.20%)	3,347	0 (0.00%)	1,297	433 (12.94%)	17470%
186.crafty	30,777	44,139,556	0 (0.00%)	44,108,779	4,926,267 (11.16%)	1,077,346	45 (0.15%)	1,046,614	78,785 (7.31%)	3997%
197.parser	15,196	32,905,403	0 (0.00%)	32,890,207	29,355,388 (89.21%)	659,867	2 (0.01%)	644,673	535,291 (81.12%)	4887%
252.eon	4,401	994,655	0 (0.00%)	990,264	974,925 (98.02%)	28,855	0 (0.00%)	24,454	22,538 (78.11%)	3347%
253.perlbmk	57,507	102,068,477	0 (0.00%)	100,349,485	94,603,019 (92.69%)	5,389,973	130 (0.23%)	5,363,373	4,461,094 (82.77%)	1794%
254.gap	7,935	10,611,636	0 (0.00%)	10,603,701	9,981,368 (94.06%)	205,200	41 (0.52%)	197,306	152,470 (74.30%)	5071%
255.vortex	29,971	265,981,817	0 (0.00%)	265,951,846	238,479,881 (89.66%)	2,159,444	98 (0.33%)	2,129,473	1,385,953 (64.10%)	12217%
256.bzip2	4,306	2,466,876	0 (0.00%)	2,462,570	708,163 (28.71%)	13,917	10 (0.23%)	9,621	1,509 (10.84%)	17626%
300.twolf	16,710	44,735,257	0 (0.00%)	44,718,440	33,741,198 (75.42%)	2,285,090	56 (0.34%)	2,268,436	1,678,383 (73.45%)	1858%
Avg.	16,861	49,414,683	0 (0.00%)	49,246,769	40,008,101 (65.47%)	1,128,352	36 (0.19%)	1,114,316	787,834 (50.80%)	7477%
176.gcc*	435,692	N/A	N/A	N/A	N/A	692M	498 (0.11%)	692M	79.43%	N/A
BAP-VSA ² on 181.mcf		# Found: 23,068	# Miss: 0 (0.00%)	# Extra: 21,018	# MisTyped: 12,533 (54.33%)	Reduce: 589%				

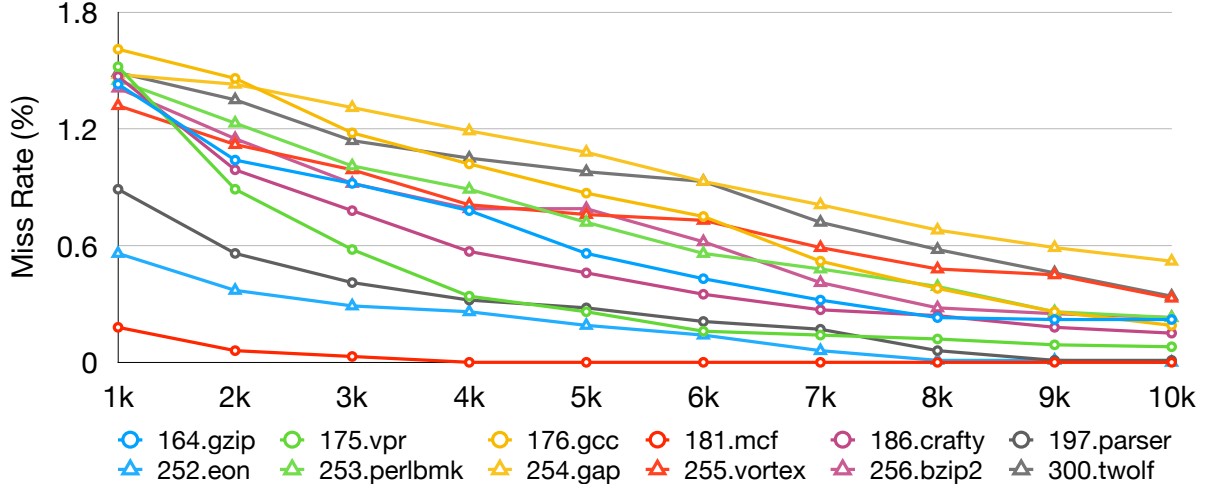


Figure 2.15. Effect of sampling

BAP-VSA). Note that it is intractable to acquire the ground truth of program dependencies, even with source code (due to various reasons such as aliasing and loops). Therefore, we use two methods to evaluate the quality of detected dependencies. First, we run the programs with the inputs provided by SPEC and use the observed dependencies as *reference*. Any dependence detected by reference executions but not by the analysis tools is called a *missing* dependence (or a false negative). Any dependence detected by the tools but not observed during reference executions is called an *extra* dependence. Note that the provided inputs achieve 81% code coverage for the SPECINT2000 benchmarks. In addition, we implemented a static type checker to validate if the source and the destination of a (detected) dependence have the same type. The checker is implemented as an LLVM pass, which propagates symbol information to individual instructions, registers and memory locations. As such, we can obtain the type of each binary operation and its operands. Note that such information is much richer than the debugging information generated during compilation. Any dependence whose source and destination have different types is considered a *mis-typed* dependence, which is most likely to be a bogus dependence. For fair comparison, we set a fixed timeout of 12 hours for each program.

Result Summary. Table 2.6 shows the result summary. Column 2 denotes the number of dependencies observed in the reference execution, columns 3-6 and 7-10 report the number

of reported dependencies, the missing ones, the extra ones and the mis-typed ones for Alto and BDA, respectively. Column 11 shows the reduction of the reported dependencies by BDA from Alto (e.g., the reduction of `181.mcf` from Alto is $(588076-3347)/3347=17470\%$). N/A in the table means that the tool times out and hence its analysis result is not available. Note that BAP-VSA only handles `181.mcf`, we list the result separately on the bottom. We have doubled the execution time for other programs but the analysis still cannot terminate. Further inspection shows that when the value set of an address operand is substantially inflated, which happens a lot in practice, each write through the address operand incurs very expensive updates for a very large number of abstract locations.

Observe that although Alto does not have any missing dependence, the number of reported dependence is very large (due to its conservativeness) and 65.47% of which are mis-typed. Such substantial bogus dependences hinder its use in practice. In comparison, the dependences reported by BDA are 75 times smaller, at the prices of a negligible missing rate (0.19%). Note that although in some cases the mis-typed rate of BDA is only slightly lower than that of Alto (e.g., `197.parser`), the absolute number of mis-typed dependences is much smaller. We argue the results by our tool are more useful in practice. We should note that the analysis of `176.gcc` is very expensive due to its complexity. As such, Alto could not finish in time. Compared to VSA, BDA reports 589% fewer dependences with a much smaller number of mistyped dependences (433 versus 12533).

We also study the reasons of missing dependences and mis-typed dependences. We find that missing dependences are mainly due to loop paths difficult to cover statically. Consider the code snippet from `164.gzip` in Figure 2.18a, BDA misses the dependence from line 6 to line 4 regarding the suffix of `dst` copied from `msg`. The reason is that BDA only iterates loop (lines 5-6) for a small number of times, which allows it to detect the dependence from line 6 to line 3 regarding the prefix of `dst`, but not the suffix. Mis-type dependences are mainly due to the fact that BDA does not model path feasibility when predicates are dependent on external inputs. As such, bogus dependences are introduced along infeasible paths. We want to point out that the same limitation applies to all analyses that do not fully model path feasibility (e.g., data-flow analysis). Consider the code snippet in Figure 2.18b from

Table 2.7. Effect of posterior analysis and taint tracking.

Program	original BDA			w/o analysis			w/o taint-tracking		
	All(K)	Miss	MisTyped	All(K)	Miss	MisTyped	All(K)	Miss	MisTyped
164.gzip	29	0.22%	11.92%	24	4.53%	2.28%	31	0.37%	23.49%
175.vpr	559	0.08%	61.88%	79	5.44%	43.86%	583	0.11%	67.13%
176.gcc	692(M)	0.11%	79.43%	14(M)	7.26%	35.52%	723(M)	0.10%	84.86%
181.mcf	3	0.00%	12.94%	2	1.17%	10.22%	4	0.10%	28.17%
186.crafty	1,077	0.15%	7.31%	124	1.88%	1.13%	1,114	0.14%	13.42%
197.parser	659	0.01%	81.12%	98	8.94%	62.96%	670	0.01%	84.34%
252.eon	28	0.00%	78.11%	10	1.52%	58.58%	28	0.00%	79.95%
253.perlbnk	5,389	0.23%	82.77%	636	5.35%	67.98%	5,524	0.25%	89.36%
254.gap	205	0.52%	74.30%	70	2.08%	36.73%	217	0.49%	81.17%
255.vortex	2,159	0.33%	64.10%	356	4.73%	58.36%	2,227	0.33%	67.30%
256.bzip2	13	0.23%	10.84%	10	2.90%	6.19%	15	0.46%	23.53%
300.twolf	2,252	0.34%	73.45%	294	6.21%	67.01%	2,375	0.35%	79.21%
Avg.	58,697	0.18%	53.18%	1,308	4.57%	37.56%	61,324	0.22%	60.16%

Table 2.8. Runtime overhead.

Program	Time ³ (h)				Memory (GB)
	Total	Pre Processing	Abstract Interpretation	Posterior Analysis	
164.gzip	1.59	0.15	1.13	0.31	3.8
175.vpr	6.80	0.54	2.75	3.51	21.4
176.gcc	10.06	1.63	7.54	0.89	103.3
181.mcf	0.83	0.06	0.71	0.06	1.6
186.crafty	7.39	0.36	2.47	4.56	15.6
197.parser	5.62	0.29	2.17	3.16	12.5
252.eon	5.98	0.84	3.51	1.63	5.7
253.perlbnk	11.35	0.68	4.24	6.43	73.5
254.gap	5.67	0.21	2.61	2.85	4.0
255.vortex	11.75	0.63	4.13	6.99	58.1
256.bzip2	2.32	0.18	1.27	0.87	4.2
301.twolf	11.68	0.57	3.99	7.12	47.9
Avg.	6.75	0.51	3.03	3.21	29.3

164.gzip. Along some infeasible paths, pointers `err_cnt` and `err_msg` are not allocated and hence have the NULL value, which leads to bogus dependence from line 6 to line 3.

Necessity of Posterior Analysis and Taint Tracking. We study the necessity of posterior analysis and taint tracking. Table 2.7 shows the effect of the posterior analysis by comparing the number of missing dependences when using the posterior analysis and when

²BAP-VSA took 10.9 hours and 8.3GB memory for 181.mcf. It timed out for the rest (exceeding 12 hours).

```

1  typedef struct node {int val; struct node *next} node_t;
2
3  node_t list_a[10000], list_b[10000];
4
5  int foo() { // list_a and list_b are independent
6      for (int i = 0, j = 1; i < 10000; i++, j++){
7          list_a[i].next = &(list_a[j % 10000]);
8          list_b[i].next = &(list_b[j % 10000]);}
9      list_a[input()].next->val = 0;
10     return list_b[input()].next->val;
11 }

```

Figure 2.16. Taint tracking example (simplified from 181.mcf)

simply aggregating the dependences collected in individual samples (0.18% versus 4.57%). We also report the total dependences. Observe that the posterior analysis produces much more dependences in total. Due to the lack of ground-truth, it is difficult to infer how many are true dependences. However, the comparison of mis-typed dependences (53.18% versus 37.56%) demonstrates the posterior analysis substantially reduces the false negative rate while only incurring a relatively modest growth of false positives (compared to the explosion incurred in Alto and VSA). Table 2.7 shows the effects of taint tracking as well. Observe that the comparisons of missing and mis-typed dependences (0.18% versus 0.22% and 53.18% versus 60.16%, respectively) indicate the necessity of taint tracking. The root cause of additional bogus dependences is that without taint tracking, constant loops are not properly interpreted (i.e., only the first a few unrolled iterations are interpreted). Consider a simplified code snippet from 181.mcf in Figure 2.16. The for-loop at line 4 is a constant loop, in which two independent node lists `list_a` and `list_b` are initialized. Without taint tracking, BDA cannot recognize it as a constant loop and hence only interprets the first a few iterations. As a result, the `next` field of the remaining `list_a` and `list_b` entries are not initialized and all have a null value. Consequently, BDA considers there is a dependence between lines 9 and 10, which is false. Since 181.mcf has many such lists and many constant initialization loops, bogus dependences are introduced between a large number of accesses through uninitialized pointers.

Table 2.9. Inferring indirect jump/call targets

Program	# Indirect Jump Edges			# Indirect Call Edges		
	IDA	Dynamic	BDA	IDA	Dynamic	BDA
164.gzip	0	0	0	0	3	3
175.vpr	49	0	49	0	1	1
176.gcc	3,628	324	3,628	25	214	853
181.mcf	0	0	0	0	1	1
186.crafty	159	38	159	0	1	1
197.parser	0	0	0	0	1	1
252.eon	17	0	17	0	183	215
253.perlbnk	1,454	229	1,454	24	243	261
254.gap	63	5	63	2	1,438	7,836
255.vortex	247	56	247	0	24	27
256.bzip2	0	0	0	0	1	1
301.twolf	17	0	17	0	1	1
Avg.	470	54	470	4	176	767

Effect of Sampling We also study the effects of having different number of samples. Figure 2.15 shows the effect of sampling. Observe that the missing rate decreases as the number of samples increases. When the number of samples reaches 10k, the missing rate is reduced to less than 0.5% for all programs and 0 for some (e.g., mcf, eon and parser). Note that the experimental results are consistent with our probabilistic analysis in Section 2.4.2.

Analysis Overhead. Table 2.8 presents the time and memory consumption of BDA in analyzing each SPEC2000 program. On average, BDA takes 6.75 hours to analyze a program, with 7%, 45% and 48% spending on the pre-processing, abstract interpretation and posterior analysis, respectively. The sampling stage takes relatively small amount of time even with the cost of dealing with large weight values. The time consumption for abstract interpretation is the sum of individual samples. The memory consumption of BDA ranges from 1.6GB to 103.3GB (29.3GB on average), depending on the complexity of the target programs. As a comparison, Alto has similar memory consumption as BDA (21.9GB vs. 22.6GB) and is 27.7% slower (8.3h vs. 6.5h) on SPECINT2000 excluding 176.gcc. We argue that since dependence graph generation is a one-time effort, the entailed overhead is reasonable.

Table 2.10. Malware behavior analysis

Malware	# Library Calls	
	Cuckoo	BDA
1a0b96488c4be390ce2072735ffb0e49	50	164
3fb857173602653861b4d0547a49b395	20	112
49c178976c50cf77db3f6234efce5eeb	23	48
5e890cb3f6cba8168d078fdede090996	28	138
6dc1f557eac7093ee9e5807385dbcb05	20	75
72afccb455faa4bc1e5f16ee67c6f915	6	81
74124dae8fdbb903bece57d5be31246b	36	203
912bca5947944fdcd09e9620d7aa8c4a	20	68
a664df72a34b863fc0a6e04c96866d4c	23	99
c38d08b904d5e1c7c798e840f1d8f1ee	34	151
c63cef04d931d8171d0c40b7521855e9	20	81
dc4db38f6d3c1e751dcf06bea072ba9c	20	77
Avg.	25	108

2.7.3 Applications

We evaluate BDA in two downstream analysis, one is to identify indirect control flow transfer targets (conducted on the SPEC programs), the other is to disclose hidden malware behaviors (conducted on 12 recent malware samples).

Inferring Indirect Control Transfer Targets. With program dependences, we can infer the potential targets of an indirect jump/call instruction by backward slicing from its target register. Table 2.9 shows the results. For comparison, we also present the analysis result of IDA and the indirect targets observed when running with inputs provided in SPEC. Observe that BDA performs as good as IDA in inferring indirect jump targets and substantially outperforms IDA in inferring indirect call targets (4 found by IDA on average versus 767 found by BDA). We should note that indirect jump targets are easier to infer than indirect call targets, as indirect jumps are always intra-procedural and have fixed patterns when they are generated by mainstream compilers. IDA leverages such patterns whereas we leverage dependences. None of the observed call targets is missed by BDA. In addition, the set of indirect call targets reported by BDA is comparable to those reported in [32], a very

```

1  int main() {
2      while (!cnc_server_connected()) sleep(5);
3      initialization();
4      .....
5  }
6  void initialization() {
7      char *dirs[10], cmd[256];
8      memcpy(dirs, rodata_41176, 0x50);
9      for (int i = 0; i < 10; i++) {
10         sprintf(cmd, "cd %s && "
11             "for a in `ls -a %s`; do >$a; done;",
12             rodata_4112A0[i], rodata_4112A0[i]);
13         system(cmd);
14     }
15 }

```

(a) simplified code

```

a lea rdi, [rbp + local_60]
b mov esi, rodata_411760 ; list of bin dirs:
   "/dev/netlink/", "/var/", ..., "/usr/"
c mov edx, 0x50
d call memcpy
e ...
f lea rdi, [rbp + local_1F0]
g lea rsi, rodata_4112A0 ; format string:
   "cd %s && for a in `ls -a %s`; do >$a; done;"
h mov rdx, [rbp + rax * 8 + local_60]
i mov rcx, rdx
j call sprintf
k ...
l lea rdi, [rbp + local_1F0]
m call system

```

(b) slicing with the dependence information

Figure 2.17. Malware case study

expensive *concrete* execution engine that forcefully executes along a large number of paths. The results demonstrate the practical use of BDA.

Exposing Malware Behaviors. Program dependence can be used to study malware behaviors. In the literature of malware analysis [21], the behavior of a malware sample is largely defined by the system calls performed by the sample, together with parameter values. With dependencies, we can perform (static) constant propagation through dependence edges to identify the parameter values of critical library functions. We also compare BDA with

```

1 char* encode_msg(char *msg, int n) {
2     char *dst = malloc((n + PREFIX_LEN));
3     strcpy(dst, PREFIX_STR);
4     strcat(dst, msg);
5     for (int i = 1; i < n + PREFIX_LEN; i++)
6         dst[i] = dst[i] ^ dst[i-1];
7     return dst;
8 }

```

(a) missing dependence

```

1 void error(char* err_msg, int *err_cnt) {
2     if (!err_cnt)
3         *err_cnt += 1; // Written type: int
4         // Potentially write to address NULL
5     if (!err_msg)
6         puts(err_msg); // Read type: char
7         // Potentially read from address NULL
8 }

```

(b) mis-typed dependence

Figure 2.18. Code snippet on missing and mistyped dependence

Cuckoo, a state-of-the-art malware analysis tool. Cuckoo reports behavior on the system call level, while BDA reports on the library call level. To be comparable, we map library calls to system calls. Table 2.10 shows the results. Observe that BDA reports 3 times more hidden malicious behaviors.

Case study. We take the malware sample `a664df72a34b863fc0a6e04c96866d4c` as a case to study how our dependence analysis can help detect hidden malicious behaviors. It is a bot malware that waits for commands from a remote server. Figure 2.17a shows the simplified code of its initialization logic. In particular, it tries to connect to a remote server every 5 seconds until success (line 2), then executes the binary files stored in some pre-defined directories (e.g., `/dev/netslink`) to setup the running environment (lines 9-14). The behavior of running the binary files will not be triggered by the sandbox execution in Cuckoo, since the remote server is down. Hence, Cuckoo fails to detect such behavior. In contrast, we perform static constant propagation through dependences to extract critical library calls with *concrete* parameters. Figure 2.17b presents the static slice of `system()`

call (line 13), whose parameter depends on the invocation of the `sprintf` library function, which fills the format string buffer indicated by `rbp + local_1F0`. It further depends on the format string stored in the global memory `rodata_4112A0`, which has the value `"cd %s && for a in `ls -a %s`; do >$a; done;"`. Hence, we detect the behavior of running the binary files under pre-defined directories without executing the malware.

2.8 Summary

We propose a practical program dependence analysis for binary executables. It features a novel unbiased whole-program path sampling algorithm and per-path abstract interpretation. Under certain assumptions, our technique has probabilistic guarantee in disclosing a dependence relation. Our experiments show that our technique has substantially improved the state-of-the-art, such as value set analysis. It also improves performance of downstream applications in indirect call target identification and malware behavior analysis.

3. PROBABILISTIC INFERENCE: RECOVERING VARIABLES AND DATA STRUCTURES

Recovering variables and data structure information from stripped binary is a prominent challenge in binary program analysis. While various state-of-the-art techniques are effective in specific settings, such effectiveness may not generalize. This is mainly because the problem is inherently uncertain due to the information loss in compilation. Most existing techniques are deterministic and lack a systematic way of handling such uncertainty. We propose a novel probabilistic technique for variable and structure recovery. Random variables are introduced to denote the likelihood of an abstract memory location having various types and structural properties such as being a field of some data structure. These random variables are connected through probabilistic constraints derived through program analysis. Solving these constraints produces the posterior probabilities of the random variables, which essentially denote the recovery results. Our experiments show that our technique substantially outperforms a number of state-of-the-art systems, including IDA, Ghidra, Angr, and Howard. Our case studies demonstrate the recovered information improves binary code hardening and binary decompilation.

3.1 Introduction

A prominent challenge in binary program analysis is to recognize variables, derive their types, and identify complex array and data structure definitions. Such information is lost during compilation, that is, variables and data structure fields are translated to plain registers and memory locations without any structural or type information. Variable accesses, including those for both simple global scalar variables and complex stack/heap data structure fields with a long reference path (e.g., `a.b.c.d`), are often uniformly compiled to dereferences of some registers that hold a computed address. Recovering the missing variable and structure information is of importance for software security. Such information can be used to guide vulnerability detection [33], legacy code hardening (e.g., adding bound checks) [34–37], executable code patching (i.e., applying an existing security patch to an executable) [38], and

decompilation (to understand hidden program behaviors) [15, 39, 40]. It is also a key step in any non-trivial binary rewriting, such as binary debloating to reduce attack surface [41].

Most binary analysis platforms have the functionalities of variable recovery and some support of structure recovery, i.e., array, struct, and class recovery. Many of them, including the most widely used IDA platform [15], hard-code a set of reverse engineering rules that are effective in certain scenarios (e.g., for binaries generated by some compilers). However, they are usually not general enough because modern compilers are diverse and feature aggressive optimizations, which may violate many instruction patterns that these rules rely on. A number of systems, including Ghidra [39], Angr [42], and TIE [43], make use of static program analysis, such as data-flow analysis and abstract interpretation, to identify variables and infer types. However, their underlying static analysis is often not sufficiently accurate. For example, many rely on *Value Set Analysis* (VSA) [19] to derive the points-to relations at the binary level. However, VSA is known to produce a lot of bogus information, reporting many memory accesses potentially aliased with almost the entire address space. Some techniques such as REWARDS [33] and Howard [44] rely on dynamic analysis to achieve better accuracy. They need high quality inputs to reach good coverage. Such inputs may not be feasible in security applications. In addition, as compilation is lossy, variable and structure recovery is inevitably uncertain. Such uncertainty often yields contradicting results. For instance, many techniques rely on specific instruction patterns of loading base address to recognize a data structure. However, such patterns may appear in code snippets that do not access data structure at all (just by chance). Existing techniques lack a systematic way of dealing with such uncertainty.

We observe that there are a large number of hints of various kinds that can be collected to guide variable and structure recovery, many of them have not been fully leveraged by existing techniques, due to both the difficulty of precluding bogus hints and the lack of a systematic way of integrating them in the presence of uncertainty. For example, some of such hints include: two objects of the same class often go through similar data-flow; two objects of the same class may have direct data-flow between their corresponding fields (due to object copying). However, leveraging such hints requires identifying precise data-flow, which is difficult, and aggregating them when there is uncertainty.

In this chapter, we propose a probabilistic variable and data structure recovery technique. It extends a recent binary abstract interpretation infrastructure BDA that has better scalability and accuracy, to collect a large set of basic behavioral properties of the subject binary, such as its memory access patterns, data-flow, and points-to relations. For each (abstract) memory location, i.e., a potential variable/data-structure-field, a set of random variables are introduced to denote its possible *primitive types* (e.g., int, long, and pointer) and its *structural properties* (e.g., being a field of some data structure or an element of some array). These random variables are correlated through the hints collected by program analysis. For example, two memory locations may be two elements of a same array if they are accessed by the same instruction. This hint can be encoded as a probabilistic constraint involving the random variables for the two memory locations. Note that although such hints are uncertain, the introduction of random variables and probabilistic constraints naturally models the uncertainty. Intuitively, a random variable may be involved in multiple hints and hence its probability is constrained by all those hints. All these probabilistic constraints are resolved together to derive the posterior distribution. We develop a customized iterative probabilistic constraint solving algorithm. It features the capabilities of handling a large number of random variables, constraints, and the need of updating the constraints on-the-fly (e.g., when disclosing a new array). It also features optimizations that leverage the domain specific modular characteristics of programs.

Our contributions are summarized as follows.

- We propose a novel probabilistic variable and data structure recovery technique that is capable of handling the inherent uncertainty of the problem.
- We develop a set of probabilistic inference rules that are capable of aggregating in-depth program behavioral properties to achieve precision and good coverage in recovery results.
- We develop an iterative and optimized probabilistic constraint solving technique that handles the challenges for probabilistic inference in program analysis context.

- We develop a prototype OSPREY (*recQuery of variable and data Structure by Proba-
bilistic analysis for strippEd binarY*) [45]. We compare its performance with a number of state-of-the-art techniques, including Ghidra, IDA, Angr, and Howard, on two sets of benchmarks collected from the literature [43, 44]. Our results show that OSPREY outperforms them by 20.41%-56.78% in terms of precision and 11.89%-50.62% in terms of recall. For complex variables (arrays and data structures), our improvement is 6.96%-89.05% (precision) and 46.45%-74.02% (recall). We also conduct two case studies: using our recovered information to (1) improve decompilation of IDA and (2) harden stripped binaries.

3.2 Motivation

In this section, we use an example to illustrate the limitations of existing techniques and motivate our technique. Figure 3.1a presents the source code of a function `huft_build` in *gzip* (lines 8-15). It is substantially simplified for the illustration purpose. We also introduce a crafted `main()` function (lines 5-7) which uses a predicate over a random number to represent that the likelihood of reaching the function through random test input generation is low (line 6). Figure 3.1b presents the corresponding assemble code, and Figure 3.1c shows part of the memory layout of the variables. In the source code, lines 1-4 define a structure `elem_t` consisting of two fields `x` and `y`; inside the function, line 9 declares `p` as a pointer to `elem_t`, and `v` as a stack-inlined `elem_t`; the conditional at line 10 has two branches, with the true branch setting `p` to the address of `v` and the false branch allocating a piece of heap memory to `p` (line 13), and storing `v` to the allocated space (line 14); and finally, line 16 outputs `p->x` and `p->y`.

After compilation, global variables are denoted by constant addresses and local variables are translated to offsets on stack frames. For example, the definitions of `v.x` and `v.y` at line 9 are translated to memory writes to stack offsets `rsp+0x8` and `rsp+0x10` (instructions [01]-[02] in Figure 3.1b, respectively). The assignment to `p` at line 11 is translated to a write to offset `rsp+0x0` at instruction [5] in Figure 3.1b. This is due to the stack memory layout shown on the left of Figure 3.1c. Observe that from the assembly code the types of these stack

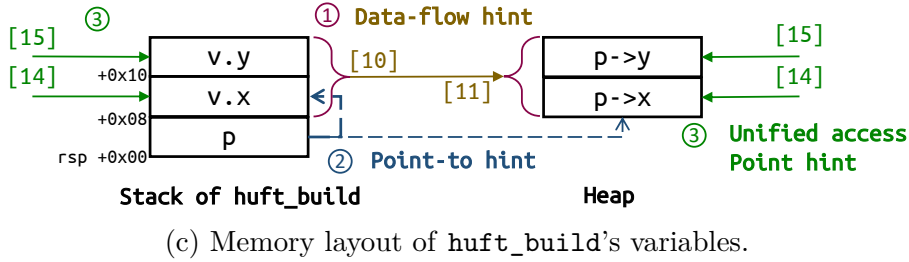
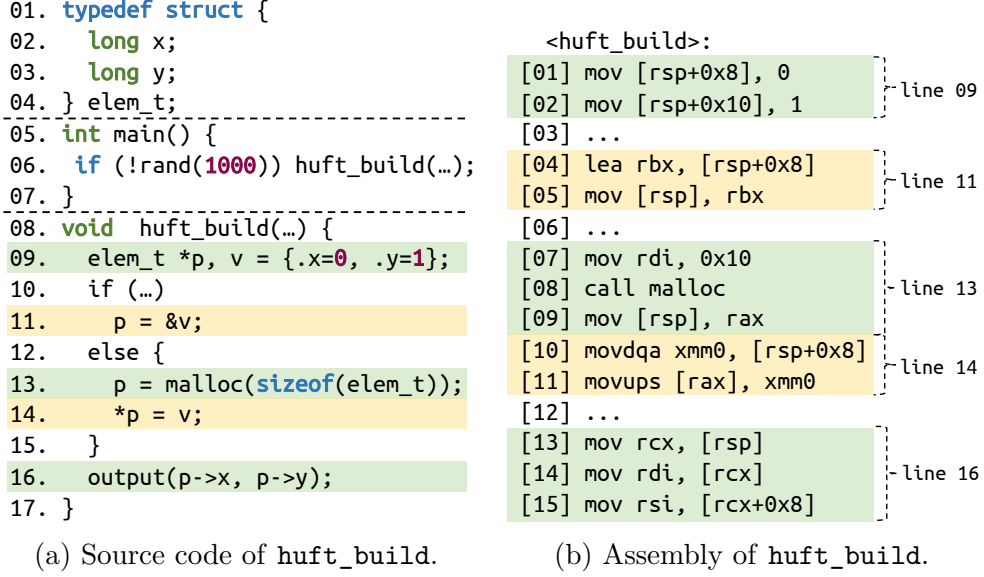


Figure 3.1. Motivation example.

offsets are unknown. It is also unclear `rsp+0x8` and `rsp+0x10` belong to a data structure while `rsp` denotes an 8-byte scalar variable. It is almost impossible to know that the heap variable stored in register `rax` at instruction [09] is of the same type as the data structure denoted by `rsp+0x8` and `rsp+0x10`. This example only represents some simple situations. In practice, there are much more difficult challenges such as nesting structures, array of structures, and arrays inside structures. In the following, we discuss how the state-of-the-art techniques and our technique perform on this example. Note that the ideal recovery result is to identify `p` as a pointer to `elem_t` while `v` is an instance of the same structure on stack, as shown in the “ground truth” column in Figure 3.2.

IDA [15] is one of the most widely-used commercial decompilation toolkits. It has the functionality of recovering variables and their types. Its recovery algorithm, which is called

<pre>typedef struct { long x; long y; } elem_t; elem_t *p; elem_t v;</pre>	<pre>typedef union { int64 u_0[2]; int128 u_1; } union_0; int128 *local_0; union_0 local_8;</pre>	<pre>typedef struct { int32 s_0[4]; } struct_0; struct_0 *local_0; int64 local_8; int64 local_10;</pre>	<pre>typedef struct { int64 s_1; int64 s_2; } struct_0; typedef struct { int64 s_1; int64 s_2; } struct_1; typedef union { struct_1 u_0; int128 u_1; } union_0; struct_0 *local_0; Union_0 *local_0; int64 local_8; int64 local_10; struct_0 local_8;</pre>
Ground Truth	IDA Pro	Ghidra	
<pre>typedef union { struct { int64 s_1; int64 s_2; } u_0; int128 u_1; } union_0; union_0 *local_0; int64 local_8; int64 local_10;</pre>	<pre>typedef struct { int64 s_1; int64 s_2; } struct_0; struct_0 *local_0; int64 local_8; int64 local_10;</pre>		
TIE ⁺ and REWARDS [*]	Howard [*]	angr	OSPREY
+ Such result requires full VSA supported. * Such results require function <huft_build> executed.			

Figure 3.2. Results of different techniques for `huft_build`

semi-naive algorithm in [46], is based on a local (intra-procedural) static analysis. It identifies absolute addresses, `rsp`-based offsets, and `rbp`-based offsets as variables or data structure fields. For example, it recognizes `rsp+0x8` (at instruction [01]) as a variable/field. In order to distinguish data structure fields from scalar variables, IDA developers hard-coded a number of code pattern matching rules. For example, they consider field accesses are performed by first loading the base address of the data structure to a register, and then adding the field offset to the register. As such, they consider all the accessed addresses that share the same base belong to a data structure. Another sample rule is that an instruction pair like the `movdqa` instruction at [10] and the `movups` instruction at [11] denotes a 128-bit packed floating-point value movement. Unfortunately, modern compilers aggressively utilize these instruction patterns to optimize code generation. In our case, the two instructions are not related to floating-point value copy but rather general data movement. As shown in Figure 3.2, IDA misidentifies `elem_t` as a union (denoted as `union_0`) of a 64-bit value array

of size two, and a monolithic field of 128-bit. The data structure is recognized through the `lea` instruction at [04], which loads the base address `rsp+0x8`. However, since `rsp+0x8` and `rsp+0x10` are accessed in two manners, one accessing individual addresses as instructions [01] and [02], and the other accessing the region as a whole like instructions [10] and [11], IDA determines that it is a union. Also observe that IDA fails to recognize that variable `local_0` (i.e., the local variable at stack offset 0 corresponding to `p` in the source code) is a pointer to the data structure. In our experiment over 101 programs (Section 3.6), IDA achieves 66.88% precision and 76.29% recall.

Ghidra [39] is a state-of-the-art decompiler developed by NSA. Its algorithm is similar to IDA’s. The improvement is that Ghidra leverages a register-based data-flow analysis [47] to analyze potential base addresses that are beyond `rsp` and `rbp` registers. In our example, it identifies `rax` at instruction [09] denotes the base address of the allocated heap structure at [08] as the return value of `malloc` at [08] is implicitly stored in `rax`. This allows Ghidra to identify `local_0` (i.e., `rsp`) as a pointer to the heap data structure as shown in the “Ghidra” column in Figure 3.2. However, the data-flow analysis is limited. It does not reason about data flow through memory. Observe that the base address in `rax` is stored to `[rsp]` at instruction [09] and then loaded to `rcx` at [13]. Ghidra cannot recognize `rcx` at [13] denotes the same base address as `rax` at [09]. As a result, it cannot recognize `local_0` is pointing to the same data structure of the two stack offsets `rsp+0x8` and `rsp+0x10`. Instead, it identifies `local_0` a 32-bit value heap array of size 4 and the two stack offsets as separate scalar variables. Inspection of Ghidra’s source code indicates that Ghidra developers do not consider stack offsets as reliable base addresses (potentially due to that compiler optimizations may lead to arbitrary stack addressing) such that it does not even group the two stack offsets to a structure. This demonstrates that the intrinsic uncertainty in variable recovery leads to inevitably ad-hoc solutions. In our experiment, Ghidra achieves 69.77% precision and 76.73% recall.

TIE [43] is a static type inference technique for binary programs. It leverages a heavy-weight abstract interpretation technique called *Value Set Analysis* (VSA) [19] to reason about data-flow through memory. VSA over-approximates the set of values that may be held in registers

and memory locations such that a memory read may read the value(s) written by a memory write as long as their address registers’ value sets have overlap, meaning that the read and the write may reference the same address. Facilitated by VSA, TIE is able to determine that the access of `[rsp]` at instruction [13] may receive its value from the write at instruction [09] that represents the allocated heap region. As such, the accesses in instructions [14] and [15] allow TIE to determine that the heap structure consists of two `int64` fields, as shown in Figure 3.2. However, VSA is conservative and hence leads to a large amount of bogus data-flow. As such, existing public VSA implementations do not scale to large programs, including *gzip*. Besides, the inherent uncertainty in variable recovery and type inference often leads to contradicting results. TIE cannot rule out the bogus results and resorts to a conservative solution of retaining all of them. Assume the underlying VSA scaled to *gzip* and hence TIE could produce results for our sample function `huft_build`. TIE would observe that instructions [14] and [15] access two `int64` fields inside the heap structure. Meanwhile, it would observe that instruction [10] directly accesses a 128 bits value in the same structure. It would consider the structure may contain just a monolithic field. To cope with the contradiction, TIE simply declares a `union` to aggregate the results, as shown in Figure 3.2. Note that since TIE is not available, in order to produce the presented results, we strictly followed their algorithm in the paper. Finally, as commented by some of the TIE authors in [40], TIE does not support recursive types, although they are widely used (e.g., in linked lists and binary trees). For example, “`struct s {int a; struct s *next}`” would be recovered as “`struct s {int a; void *next}`” at best.

REWARDS [33] is a binary variable recovery and type inference technique based on dynamic analysis. Through dynamic tainting, it precisely tracks data-flow through registers and memory such that base-addresses and field accesses can be recognized with high accuracy. However, its effectiveness hinges on the availability of high quality inputs, which may not be true in many security applications. Theoretical, one could use fuzzing [10, 48–51] or symbolic execution [52–56] to generate such inputs. However, most these techniques are driven by a more-or-less random path exploration algorithm whose goal is to achieve new code coverage. In our example, we use a random function (line 6) to denote the small likeli-

hood of function `huft_build()` being covered by path exploration. If functions, code blocks, and program paths are not covered, the related data-flow and hence the corresponding variable/field accesses cannot be recovered by REWARDS. Similar to TIE, REWARDS cannot deal with uncertainty. In Figure 3.2, if we assume the function has all its paths covered, REWARDS would generate the same undesirable result as TIE.

Howard [44] is also dynamic analysis based. It improves REWARDS using heuristics to resolve conflicting results. For example, it favors data structures with fields over monolithic scalar variables. Thus, the 128-bit floating-point value copy at instruction [10] is ignored by Howard in light of the field accesses at instructions [14] and [15], leading to the correctly recovered type for the heap structure. However, Howard employs a number of heuristics to tolerate the various code patterns induced by compiler optimizations. For example, it does not consider `rsp+0x8` as a valid base address. As such, *Howard* mis-classifies offsets `rsp+0x8` and `rsp+0x10` as two separate variables `local_8` and `local_10` as in Figure 3.2. This illustrates the difficulty of devising generally applicable deterministic heuristics due to the complex behaviors of modern compilers. A heuristic rule being general in one case may become too strict in another case.

Angr [42] is a state-of-the-art open-sourced binary analysis infrastructure, which is widely used in academia and industry. Its variable recovery does not rely on either static or dynamic analysis. Instead, it leverages its built-in concolic execution engine which combines symbolic execution [57] and forced execution [32, 58] to recover variables and their data-flow. Despite the more precise basic information (e.g., data-flow), Angr’s variable recovery and type inference are not as aggressive as a few other techniques, especially in the presence of conflicting results. Hence, in Figure 3.2, the current implementation of Angr cannot recognize the structure on the heap or on the stack. In our experiment (Section 3.6), Angr achieves 33.04% precision and 59.27% recall.

3.2.1 Our Technique

Observations. From the above discussion, we observe that compilation and code generation is a lossy procedure, whose reverse function is inherently uncertain. It is hence very difficult

to define generally applicable rules to recover variables. In addition, the underlying analysis plays a critical role. These analysis have different trade-offs in accuracy, scalability, and the demand of high quality inputs.

Insights. The first insight is that *while existing techniques mostly focus on memory access patterns (i.e., base addresses and offset values) to identify structures, there are many other program behaviors that can serve as hints to recover data structures*. For example, they include the following. The first is called *data-flow hint*. In Figure 3.1c, there is direct data-flow from v to $*p$, denoted by the brown arrow ①, due to the copy at instructions [10] and [11]. It implies that the two memory regions may be of the same complex type. The second kind of hints originates from points-to relations, called *points-to hint*. As blue arrows ② in Figure 3.1c indicate, variable p may point to both v and $*p$, suggesting that they are of the same type. The third kind of hint is called *unified access point*. The green arrows ③ mean that instruction [14] accesses both $v.x$ and $p \rightarrow x$, while instruction [15] accesses both $v.y$ and $p \rightarrow y$. Instructions [14] and [15] are likely unified access points to fields of the same data structure.

The second insight is that *the various kinds of hints in variable/structure recovery can be integrated in a more organic manner using probabilistic inference [59]*. Instead of making a deterministic call of the type of a memory region, depending on the number of hints collected, we compute the probabilities for the memory region having various possible types. This requires developing a set of probabilistic inference rules specific to variable recovery. In our example, the float-point instructions at instructions [10] and [11] cause a conflict, which is suppressed by the large number of other hints (e.g., ①, ②, and ③ in Figure 3.1c) in probabilistic analysis.

To realize the above two insights, a critical challenge is to precisely identify data-flow and points-to relations. The recent advance made by BDA makes this feasible.

Our Technique. For each memory location, we introduce multiple random variables to denote the probabilities of possible types of the memory location. We construct the set of possible types and compute the probabilities for these random variables as follows. Specifically, OSPREY extends BDA to compute valuable program properties (introduction to BDA and

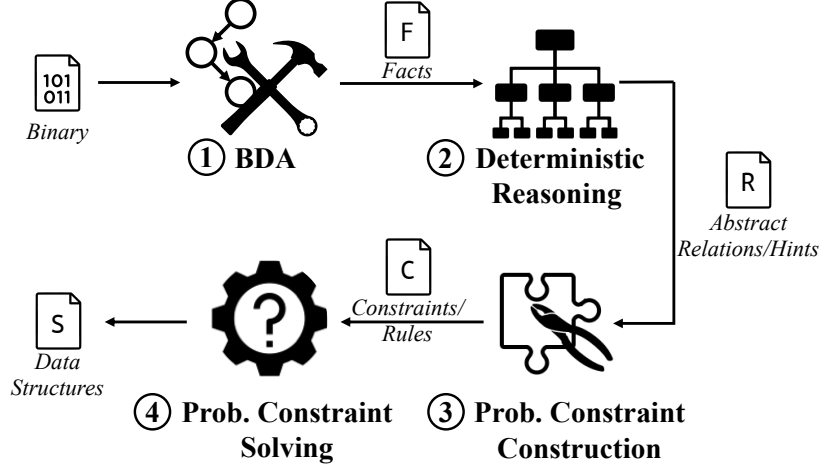


Figure 3.3. System design

our extension can be found in Sections 3.3 and 3.4), including memory access patterns, data-flow through register and memory, points-to, heap usage, and so on. These program properties are regarded as basic facts, each of which has a prior probability representing its implication of typing and structural properties. For example in Figure 3.1c, the *points-to hint* ② that `p` may point to both `v` and `*p` indicates a large prior probability that `v` and `*p` are of the same type. After collecting all the hints with their probabilities, OSPREY performs probabilistic inference to propagate and aggregate these hints, and derive the posterior marginal probabilities that indicate the probable variables, types, and data structure declarations. For instance, in Figure 3.2, the likelihood of `v` (or `local_8`) being a stack based structure is much higher than that of two separated `int64s` (0.7 `v/s` 0.3). The likelihood of `p` (or `local_0`) being a pointer to a structure is much higher than being a pointer to a union (0.9 `v/s` 0.1). This aligns perfectly with the ideal result. Our experiments show that if we only report the most probable ones, our technique can achieve 90.18% precision and 88.62% recall, and 89.05% precision and 74.02% recall for complex variables (e.g., `struct`), substantially outperforming other existing techniques.

3.3 Design Overview

Figure 3.3 shows the workflow of OSPREY. Given a stripped binary, BDA is first used to collect basic analysis facts of the binary (e.g., data-flow and points-to). These basic facts are then first processed by a deterministic reasoning step ②. For example, access/data-flow patterns can be extracted and compared to form hints. The resulted abstract relations/hints then go through the probabilistic constraint construction step ③, where predicates describing structural and type properties of individual memory chunks are introduced (e.g., whether a memory chunk denotes a field starting at some memory address), each denoted by a random variable. Here a memory chunk is a smallest memory unit accessed by some instruction. A set of inference rules are introduced to describe the correlations across these random variables. As such, a random variable is constrained in multiple ways (by various hints). In step ④, these constraints/rules are transformed to a probabilistic graph model. A customized inference algorithm (developed from scratch) is then used to resolve these probabilistic constraints to produce the posterior probabilities. Different from most existing probabilistic inference algorithms, our algorithm is iterative to deal with on-the-fly changes of the constraints, which are inevitable due to the nature of our problem. For example, finding a new likely array leads to introduction of new predicates denoting its properties and requires re-inference. Our algorithm is also optimized as most existing inference engines cannot deal with the large number of random variables in our context. Our optimization leverages the modular characteristics commonly seen in programs and program analysis. Finally, the most probable type and structural predicates are reported and further processed to generate the final variable, type, and structure declarations.

3.4 Deterministic Reasoning

Before probabilistic inference, our technique performs deterministic reasoning, through which analysis facts are collected and processed to derive a set of relations and hints. Such information provides the needed abstraction so that the later probabilistic inference, which is sensitive to problem scale, does not have to be performed on the low-level facts.

$f \in \langle \text{Function} \rangle ::= \text{Int}_{64}$	$o \in \langle \text{Offset} \rangle ::= \text{Int}_{64}$
$i \in \langle \text{Instruction} \rangle ::= \text{Int}_{64}$	$k \in \langle \text{Constant} \rangle ::= \text{Int}_{64}$
$s \in \langle \text{Size} \rangle ::= \text{UInt}_{64}$	$r \in \langle \text{MemRegion: MR} \rangle ::= \mathcal{G} \mathcal{H}_i \mathcal{S}_f$
$a \in \langle \text{MemAddress: MA} \rangle ::= \langle r, o \rangle$	$v \in \langle \text{MemChunk: MC} \rangle ::= \langle a, s \rangle$

Figure 3.4. Definitions

Primitive Analysis Facts		
F_{01}	$\text{Access}(i, v, k)$: Memory chunk v was accessed by instruction i for $k > 0$ times during sampling.
F_{02}	$\text{BaseAddr}(i, v, a)$: Instruction i has accessed memory chunk v with base address a during sampling.
F_{03}	$\text{MemCopy}(v_s, v_d)$: The value loaded from v_s was stored to v_d directly, or indirectly via register copying in the middle during sampling.
F_{04}	$\text{PointsTo}(v, a)$: Memory chunk v stored an address a during sampling.
F_{05}	$\text{MallocSize}(i, s)$: The <code>malloc</code> function call at instruction i requested s bytes.
F_{06}	$\text{MayArray}(a, k, s)$: There may be an array with k elements, each s bytes, starting from address a .

Figure 3.5. Primitive analysis facts

Definitions. As shown in Figure 3.4, we use f to denote a **function**, which is essentially a 64-bit integer denoting the function’s entry point, o to denote an **offset**, i to denote an **instruction**, which is essentially a 64-bit integer representing the starting address of the instruction, and s to denote a **size**.

The memory space is partitioned to three distinct regions: *global*, *stack*, and *heap*. The global region, denoted as \mathcal{G} , stands for the space holding all the initialized and uninitialized global data. A stack frame or a heap-allocated block constitutes a region as well.

Here, we assume that a binary is correctly disassembled and function entries are properly identified such that the correctness of memory partition can be guaranteed. Although these are very challenging tasks, addressing them is beyond the scope of our discussion. There are existing techniques [38, 60–64] that particularly focus on these problems. A stack region for

Helper Functions			
H_{01}	$SameRegion(a_1, a_2) : \text{Bool}$	$::= a_1.r = a_2.r$	e.g., $SameRegion(\langle S, 0 \rangle, \langle S, 8 \rangle) : \text{true}$
H_{02}	$Offset(a_1, a_2) : \text{Size} \cup \{\infty\}$	$::= SameRegion(a_1, a_2) ? a_1.o - a_2.o : \infty$	e.g., $Offset(\langle S, 8 \rangle, \langle S, 0 \rangle) = 8$
H_{03}	$AdjacentChunk(v_1, v_2)^* : \text{Bool}$	$::= Offset(v_2.a, v_1.a) = v_1.s$	e.g., $AdjacentChunk(\langle \langle S, 0 \rangle, 8 \rangle, \langle \langle S, 8 \rangle, 1 \rangle) = \text{true}$
H_{04}	$OverlappingChunk(v_1, v_2)^* : \text{Bool}$	$::= Offset(v_2.a, v_1.a) < v_1.s$	e.g., $OverlappingChunk(\langle \langle S, 0 \rangle, 8 \rangle, \langle \langle S, 4 \rangle, 1 \rangle) = \text{true}$
H_{05}	$AddrDifferenceGCD(a_1, a_2, \dots, a_n)^+ : \text{Size}$	$::= \gcd(\{Offset(a_{k+1}, a_k) \mid 0 \leq k < n\})$	e.g., $AddrDifferenceGCD(\langle S, 0 \rangle, \langle S, 8 \rangle, \langle S, 32 \rangle) = 8$
	$SizeDifferenceGCD(s_1, s_2, \dots, s_n)^+ : \text{Size}$	$::= \gcd(\{s_{k+1} - s_k \mid 0 \leq k < n\})$	e.g., $SizeDifferenceGCD(12, 20, 36, 72) = 8$
H_{06}	$MallocatedSizes(i) : \mathcal{P}(\text{Size})$	$::= \{s_k \in \text{Size} \mid MallocatedSize(i, s_k)\}$	in ascendant order
H_{07}	$AccessedAdrsInRegion(i, r) : \mathcal{P}(\text{MC})$	$::= \{v.a \in \text{MA} \mid (v.a.r = r) \wedge Access(i, v)\}$	

*Assuming $Offset(v_2.a, v_1.a) \geq 0$ without losing generality

⁺Assuming $\forall k \in [0, n), Offset(a_{k+1}, a_k) \geq 0$ and $s_{k+1} - s_k \geq 0$ without losing generality.

Figure 3.6. Helper functions

a function f , denoted as \mathcal{S}_f , models the stack frame that holds local variables/structures for f . A heap region allocated at an instruction i is denoted as \mathcal{H}_i . A memory region r could be any of the three kinds. A memory address a is represented as $\langle r, o \rangle$, in which r stands for the region a belongs to and o for a 's offset relative to the base of the region. A *memory chunk*, which is a term we inherit from VSA [19], denotes a variable-like smallest memory unit that is ever visited by some instruction. It is represented as $\langle a, s \rangle$ where a models the starting address of the unit and s its size. It may correspond to a scalar variable, a data structure field, or an array element of some primitive type.

Consider the assemble code at instruction [11], “`movups [rax], xmm0`”, in Figure 3.1b. As register `rax` acquires its value $a = \langle r = \mathcal{H}_{08}, o = 0 \rangle$ from instruction [08], the `movups` instruction accesses a 16-byte variable-like memory chunk $v = \langle a = \langle \mathcal{H}_{08}, 0 \rangle, s = 16 \rangle$.

Primitive Analysis Facts Collected by BDA. As the first step, we extend BDA to collect a set of basic facts. Recall that BDA is a per-path abstract interpretation technique

Deterministic Inference Rules		
R_{01}	$Accessed(i, v)$	$\vdash Access(i, v, \hat{k})$
R_{02}	$Accessed(v)$	$\vdash Accessed(\hat{i}, v)$
R_{03}	$AccessSingleChunk(i, r)$	$\vdash AccessedAddrsInRegion(i, r) = 1$
R_{04}	$AccessMultiChunks(i, r)$	$\vdash AccessAddrsInRegion(i, r) > 1$
R_{05}	$HiAddrAccessed(i, r, a_h)$	$\vdash a_h = \max(AccessAddrsInRegion(i, r))$
R_{06}	$LoAddrAccessed(i, r, a_l)$	$\vdash a_l = \min(AccessedAddrsInRegion(i, r))$
R_{07}	$MostFreqAddrAccessed(i, r, a_f, k)$	$\vdash k = \max(\{k_t Access(i, v, k_t)\}) \wedge Access(i, v, k) \wedge v.a = a_f$
R_{08}	$ConstantAllocSize(i, s)$	$\vdash (MallocatedSizes(i) = 1) \wedge (s \in MallocatedSizes(i))$
R_{09}	$AllocUnit(i, s)$	$\vdash (MallocatedSizes(i) > 1) \wedge$ $(SizeDifferenceGCD(MallocatedSizes(i)) = s)$
R_{10}	$DataFlowHint(a_s, a_d, s)$	$\vdash a_s = v_s.a \wedge a_d = v_d.a \wedge (Offset(v'_s.a, a_s) = Offset(v'_d.a, a_d) = s) \wedge$ $SameRegion(a_s, v'_s.a) \wedge SameRegion(a_d, v'_d.a) \wedge MemCopy(v_s, v_d) \wedge MemCopy(v'_s, v'_d)$
R_{11}	$UnifiedAccessPntHint(a_s, a_d, s)$	$\vdash a_s = v_s.a \wedge a_d = v_d.a \wedge (Offset(v'_s.a, a_s) = Offset(v'_d.a, a_d) = s) \wedge$ $SameRegion(a_s, v'_s.a) \wedge SameRegion(a_d, v'_d.a) \wedge Accessed(i_1, v_s) \wedge Accessed(i_1, v_d) \wedge$ $Accessed(i_2, v'_s) \wedge Accessed(i_2, v'_d)$
R_{12}	$PointsToHint(a_s, a_d, s)$	$\vdash a_s = v_s.a \wedge a_d = v_d.a \wedge (Offset(v'_s.a, a_s) = Offset(v'_d.a, a_d) = s) \wedge$ $SameRegion(a_s, v'_s.a) \wedge SameRegion(a_d, v'_d.a) \wedge BaseAddr(-, v'_s, a_s) \wedge BaseAddr(-, v'_d, a_d) \wedge$ $PointsTo(v_x, a_s) \wedge PointsTo(v_x, a_d)$

Figure 3.7. Deterministic reasoning rules

driven by path sampling. It uses precise symbolic values (i.e., without approximation) and interprets individual paths separately. One can consider that BDA is analogous to executing the subject binary on an abstract domain. It does not need to merges values across paths like other abstract interpretation techniques (e.g., VSA), so the abstract domain is precise instead of approximate. We collect six types of facts such as memory access behaviors and points-to relations, as presented in Figure 3.5. Specifically, $Access(i, v, k)$ $[F_{01}]$ states that instruction i accessed a memory chunk v for k times during the sample runs. By precisely tracking data-flow through both registers and memory, BDA can determine the base address of all offsetting operations. In particular, it looks for data-flow paths that starts by loading an address to a register, which is further copied to other registers or memory chunks, incremented by constant offsets, and eventually dereferenced. $BaseAddr(i, v, a)$ $[F_{02}]$

denotes that i accessed a memory chunk v whose base address is a . $MemCopy(v_s, v_d)$ $[F_{03}]$ states that chunk v_s was copied to v_d . It is abstracted from a data-flow path from a memory read to a memory write, with possible register copies in the middle. $PointsTo(v, a)$ $[F_{04}]$ states that an address value a was ever stored to v . Intuitively, one can consider v a pointer pointing to a . $MallocedSize(i, s)$ $[F_{05}]$ records that a memory allocation function invocation i ever requested size s . $MayArray(a, k, s)$ $[F_{06}]$ denotes that a may start an array of k elements, each with size s . Similar to Ghidra and IDA, these array-related hints are collected via heuristics, e.g., by looking at the arguments of `calloc` library call. We will show later that we have more advanced inference rules for arrays. $MayArray$ only denotes the direct hints.

Consider the motivation example in Figure 3.1b and assume the function `huft_build` was sampled 10 times. Thus, instruction [01] was executed 10 times. As `rsp` stores the base address of region $\mathcal{S}_{\text{huft_build}}$, we have $Access(01, \langle \langle \mathcal{S}_{\text{huft_build}}, 8 \rangle, 8 \rangle, 10)$ for the first instruction. At instruction [08], `malloc` is called to request 16 bytes of memory, represented by $MallocedSize(08, 16)$. After that, `malloc` returns the base address of heap region \mathcal{H}_{08} and stores it to `rax`. Instruction [09] further stores this address to `[rsp]`. Hence we get $PointsTo(\langle \langle \mathcal{S}_{\text{huft_build}}, 0 \rangle, 8 \rangle, \langle \mathcal{H}_{08}, 0 \rangle)$. Instructions [10] and [11] copy value from `rsp+0x8` to `rax`, generating $MemCopy(\langle \langle \mathcal{S}_{\text{huft_build}}, 8 \rangle, 16 \rangle, \langle \langle \mathcal{H}_{08}, 0 \rangle, 16 \rangle)$. Instruction [15] accesses `rcx+0x8` where `rcx` is the base register holding the value of $\langle \mathcal{H}_{08}, 0 \rangle$, we have $BaseAddr(15, \langle \langle \mathcal{H}_{08}, 8 \rangle, 8 \rangle, \langle \mathcal{H}_{08}, 0 \rangle)$.

Helper Functions. In Figure 3.6, we define a number of helper functions that are derived from the six kinds of basic analysis facts. These helper functions essentially derive aggregated information across a set of primitive analysis facts. They will be used in the inference rules discussed later. Specifically, $SameRegion(a_1, a_2)$ $[H_{01}]$ determines whether two memory addresses belong to the same memory region. Note that in Figure 3.6, the explanation and example for each helper function are to its right. $Offset(a_1, a_2)$ $[H_{02}]$ returns the offset between two memory addresses, which equals to the difference between their offset values if the two addresses belong to the same region, ∞ otherwise. $AdjacentChunk(v_1, v_2)$ $[H_{03}]$ determines if two memory chunks are next to each other. $AddrDifferenceGCD(a_1, \dots, a_n)$ $[H_{05}]$ returns the *greatest common divisor* (GCD) of the differences of a list of sorted addresses. $SizeDifferenceGCD(s_1, \dots, s_n)$ $[H_{05}]$ returns the GCD of the differences between a list of

sorted sizes. $MallocedSizes(i)$ $[H_{06}]$ returns the list of requested sizes from a malloc-site i . $AccessedAddrsInRegion(i, r)$ $[H_{07}]$ returns all the addresses accessed by i in region r .

Deterministic Inference Rules. The goal of deterministic inference is to derive additional relations that were not explicit. In Figure 3.7, we present the inference rules in the following format.

$$T :- P_1 \wedge P_2 \wedge \dots \wedge P_n$$

T is the target relation and P_i is a predicate. It means that the satisfaction of predicates P_1, P_2, \dots, P_n leads to the introduction of T . Observe that no probabilities are involved.

Specifically, $Accessed(i, v)$ $[R_{01}]$ denotes if instruction i has accessed memory chunk v and $Accessed(v)$ $[R_{02}]$ denotes if v has been accessed. They are derived from the primitive fact $Access(\dots)$ $[F_{01}]$. The next two relations model the access pattern of instruction i in memory region r . $AccessSingleChunk(i, r)$ $[R_{03}]$ denotes that instruction i is always accessing only one memory chunk in region r . A typical example is an instruction writing to a constant address, e.g., instruction “`mov [0xdeadbeef], 0`”. $AccessMultiChunks(i, r)$ $[R_{04}]$, in contrast, denotes i accessed multiple chunks in r , such as an instruction in some for-loop that accesses individual elements in a memory buffer. $HiAddrAccessed(i, r, a_h)$ $[R_{05}]$ dictates that a_h is the highest address in r accessed by i . $LoAddrAccessed(i, r, a_l)$ $[R_{06}]$ is the inverse. $MostFreqAddrAccessed(i, r, a_f, k)$ $[R_{07}]$ denotes a_f is the most frequently accessed address in r by i .

The next two rules describe the allocation patterns. $ConstantAllocSize(i, s)$ $[R_{08}]$ denotes that i has only requested one size s . $AllocUnit(i, s)$ $[R_{09}]$ determines if i allocated memory of different sizes and the differences are all multiples of s .

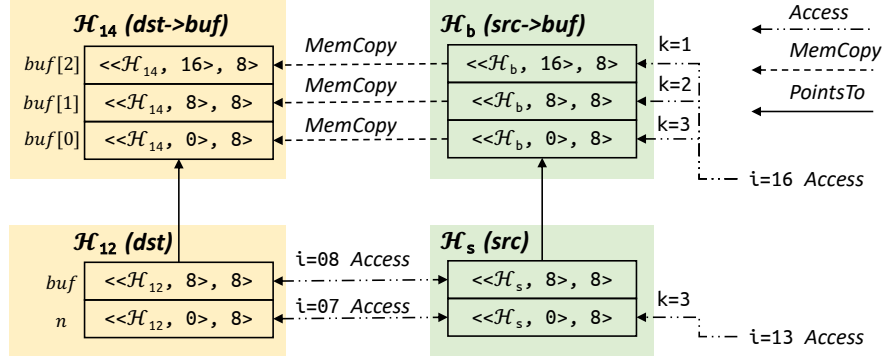
The next three rules describe the three kinds of hints (Section 3.2). $DataFlowHint(a_s, a_d, s)$ $[R_{10}]$ suggests the presence of structure if there are copies from two addresses separated by an offset (e.g., two fields from a structure) to two other respective addresses separated by the same offset. Formally, it renders true if given two addresses a_s and a_d , there are two other addresses (denoted by $v'_s.a$ and $v'_d.a$) that have the same offset from a_s and a_d , respectively, such that there are memory copies from a_s to a_d and $v'_s.a$ to $v'_d.a$. Here a_s and a_d denote two instances of the same structure. $UnifiedAccessPntHint(a_s, a_d, s)$ $[R_{11}]$ suggests

```

01. typedef struct {
02.     int n;
03.     char *buf;
04. } str_t;
05.
06. void my_print(str_t *s) {
07.     size_t n = s->n;
08.     char *buf = s->buf;
09.     write(1, buf, n);
10. }
11. str_t *my_strcpy(str_t *src) {
12.     str_t *dst = malloc(sizeof(str_t));
13.     int n = dst->n = src->n;
14.     dst->buf = malloc(sizeof(char) * n);
15.     for (int i = 0; i < n, i++)
16.         dst->buf[i] = src->buf[i];
17.     my_print(src);
18.     my_print(dst);
19.     return dst;
20. }

```

(a) Source code



(b) Memory regions (boxes), chunks (entries in box), and relations (arrows)

Figure 3.8. Example for deterministic reasoning

the presence of structure if two addresses (i.e., denoting the same field from two instances of the same structure) are accessed by a same instruction i_1 and their offsets are also accessed by another same instruction i_2 . Formally, it renders true given two addresses a_s and a_d , there are two other addresses (denoted by $v'_s.a$ and $v'_d.a$) that have the same offset from a_s and a_d , respectively, such that a_s and a_d are accessed by an instruction i_1 and $v'_s.a$ and $v'_d.a$ accessed by another instruction i_2 . $PointsToHint(a_s, a_d, s)$ $[R_{12}]$ determines a_s and a_d may denote two instances of the same structure if a_s and a_d are two base addresses for two other addresses that have the same s offset from the base, and both a_s and a_d are stored to the same pointer variable.

In Figure 3.8, we use a customized string copy function to demonstrate the deterministic reasoning procedure, with the source code in Figure 3.8a. Lines 1-4 define a **struct** **str_t** that consists of an **int** field **n** and a **char *** field **buf**, indicating the string's length and memory location, respectively. Lines 6-10 define a **my_print()** function that prints a **str_t**

structure to `stdout`. Function `my_strcpy()` copies `src` to a heap-allocated `dst` (lines 12-16), and then prints the two strings (lines 17-18). Note that we use source code to illustrate for easy understanding, while OSPREY works on stripped binaries.

Assume BDA samples `my_strcpy()` 3 times, and `src->n` equals to 1, 2, and 3, in the respective sample runs. Assume `sizeof(char)=8`. Figure 3.8b illustrates the regions (denoted by the colored boxes), the memory chunks in regions from all three runs (denoted by the entries inside the colored boxes), and the derived relations (denoted by the arrows). For example, the arrow at the lower-right corner indicates a relation $Access(13, \langle \mathcal{H}_s, 0 \rangle, 8), 3$. Observe that all the accessed fields of `src` locate in region \mathcal{H}_s , the lower green box, and all the accessed elements in `src->buf` locate in region \mathcal{H}_b , the upper green box. At line 12, function `malloc`'s parameter is always 16, leading to relation $ConstantAllocSize(12, 16)$. Expression `src->n` at line 13 only accesses a memory chunk $\langle \mathcal{H}_s, 0 \rangle$, leading to $AccessSingleChunk(13, \mathcal{H}_s)$. In contrast, from the accessed addresses `src->buf[i]` at line 16, we have $AccessMultiChunks(16, \mathcal{H}_b)$, $HiAddrAccessed(16, \mathcal{H}_b, \langle \mathcal{H}_b, 16 \rangle)$, and $LoAddrAccessed(16, \mathcal{H}_b, \langle \mathcal{H}_b, 0 \rangle)$, and $MostFreqAddrAccessed(16, \mathcal{H}_b, \langle \mathcal{H}_b, 0 \rangle, 3)$ denoting the most frequent accessed address is $\langle \mathcal{H}_b, 0 \rangle$, i.e., `src->buf[0]` (accessed three times in the three sample runs).

Consider the `my_print()` function, where line 7 accesses both $\langle \mathcal{H}_s, 0 \rangle$ and $\langle \mathcal{H}_{12}, 0 \rangle$, with \mathcal{H}_{12} the heap region allocated at 12, and line 8 accesses both $\langle \mathcal{H}_s, 8 \rangle$ and $\langle \mathcal{H}_{12}, 8 \rangle$ that have the same offset, indicating $UnifiedAccessPntHint(\langle \mathcal{H}_s, 0 \rangle, \langle \mathcal{H}_{12}, 0 \rangle, 8)$. Intuitively, the corresponding fields of two structures `dst` and `src` are accessed by the same instructions, which implies the presence of structure. Inside function `my_strcpy()`, we acquire a data-flow hint due to the copies from `src` to `dst`. Specifically, we have $DataFlowHint(\langle \mathcal{H}_b, 0 \rangle, \langle \mathcal{H}_{14}, 0 \rangle, 16)$. From the invocation interface between `my_strcpy()` and `my_print()`, we have $PointsToHint(\langle \mathcal{H}_s, 0 \rangle, \langle \mathcal{H}_{12}, 0 \rangle, 16)$ because both the base addresses of `src` and `dst` have been stored to the same function parameter of `my_print()`. \square

3.5 Probabilistic Reasoning

As discussed in Section 3.2, variable and structure recovery is a process with inherent uncertainty such that the collected hints may have contradictions due to: (1) the behavior

patterns defining hints may happen by chance, instead of reflecting the internal structure; (2) BDA’s per-path interpretation may not respect path feasibility such that infeasible behaviors may be included in the deterministic reasoning step. For example, violations of path feasibility may lead to out-of-bound buffer accesses and then bogus data-flow hints. We resort to probabilistic inference to resolve such contradictions. Intuitively, the effects of incorrect hints will be suppressed by the correct ones which are dominant. In particular, for each memory chunk v , we introduce a number of random variables to describe the type and structural properties of v . The random variables of multiple memory chunks are hence connected through the relations derived from the previous deterministic reasoning step and represented as a set of probabilistic inference rules. Each rule can be considered a probability function. They are transformed to a probabilistic graph model [65] and an inference algorithm is used to compute the posterior marginal probabilities. The most probable results are reported. Different from many existing probabilistic inference applications, where the set of inference rules are static, we have *dynamic inference rules*, meaning that rules will be updated, removed, and added on the fly based on the inference results. We hence develop an iterative and optimized inference algorithm (Section 3.5.2).

3.5.1 Probabilistic Inference Rules

Predicates and Random Variables. Figure 3.9 presents the set of predicates we introduce. They denote the typing and structural properties. Random variables are introduced to denote their instantiations on individual instructions and memory chunks, each describing the likelihood of the predicate being true. For instance, The random variable for $Scalar(\langle\langle G, 0x8043abf0 \rangle, 8 \rangle)$ denotes the likelihood that the 8-byte global memory chunk starting at $0x8043abf0$ is a scalar variable. In the remainder of the chapter, we will use the two terms predicate and random variable interchangeably. Specifically, $PrimitiveVar(v)[P_{01}]$ asserts that memory chunk v denotes a primitive variable, which is a variable without further inner structure. It could be a scalar variable, a structure field, or a primitive array element. Similarly, $PrimitiveAccess(i, v)[P_{02}]$ asserts that instruction i exclusively accesses a primitive variable v . The meanings of *UnfoldableHeap* and *FoldableHeap* will be explained in the later

P_{01}	$PrimitiveVar(v)$: v is of primitive type, e.g., char , int , and void *
P_{02}	$PrimitiveAccess(i, v)$: Instruction i accessed a primitive variable v
P_{03}	$UnfoldableHeap(i, s)$: The size of the unfoldable part of heap structure allocated at i is s
P_{04}	$FoldableHeap(i, s)$: The unit size of the foldable part of heap structure allocated at i is s
P_{05}	$HomoSegment(a_1, a_2, s)$: The two s -byte segments starting at a_1 and a_2 , respectively, are homomorphic
P_{06}	$ArrayStart(a)$: Address a is the starting address of an array
P_{07}	<u>$Scalar(v)$</u>	: Variable v is a scalar
P_{08}	<u>$Array(a_1, a_2, s)$</u>	: Memory from a_1 to a_2 belongs to an array whose element size is s -byte
P_{09}	<u>$FieldOf(v, a)$</u>	: Variable v is a field of a structure with starting address a
P_{10}	<u>$Pointer(v, a)$</u>	: Variable v is a pointer pointing to a structure denoted by a
P_{11}	<u>$IntVar(v)$ / $LongVar(v)$ / ...</u>	: Variable v is of the int / long / ... type

Figure 3.9. Predicate definitions

discussion of heap structure recovery. $HomoSegment(a_1, a_2, s)$ [P_{05}] asserts that the memory region $a_1 \sim (a_1 + s)$ and $a_2 \sim (a_2 + s)$ are homomorphic, hence likely two instances of the same structure. They are likely homomorphic when their access patterns and data-flow are similar. $ArrayStart(a)$ [P_{06}] represents a is the starting address of an array.

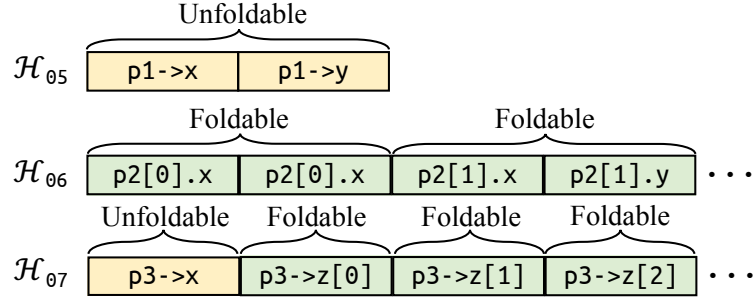
While the above predicates are auxiliary, the remaining ones (underlined in Figure 3.9) denote our final outcomes. Variables, structures and types can be directly derived from the inferred values of these predicates. In particular, $Scalar(v)$ [P_{07}] indicates v is a scalar variable (not an array or a structure). $Array(a_1, a_2, s)$ [P_{08}] represents that the memory region from a_1 to a_2 form an array of size s . $FieldOf(v, a)$ [P_{09}] asserts that v is a field of a structure starting at a . $Pointer(v, a)$ [P_{10}] asserts v is a pointer to a structure starting at a . The last few predicates assert the primitive types of variables. Note that they allow us to express the most commonly seen structural properties, including nesting structures, array of structures, and structure with array field(s). We have other predicates for unions. They are elided for discussion simplicity.

```

01. typedef struct { long x; long y; } A;
02. typedef struct { long x; long z[]; } B;
03.
04. void heap_example(size_t n, size_t m) {
05.     A *p1 = malloc(sizeof(A));
06.     A *p2 = malloc(sizeof(A) * n);
07.     B *p3 = malloc(sizeof(B) + sizeof(long [m]));
08.     ...

```

(a) Source code.



(b) Memory layout of code in Figure 3.10a.

Figure 3.10. Example to demonstrate our heap model

Example. Consider an example in Figure 3.10, with the source code in Figure 3.10a. Three types of structures are allocated on the heap. Line 5 allocates a singleton structure ($*p1$); line 6 allocates an array of the same structure ($*p2$); and line 7 allocates a structure ($*p3$) with an array. Note that the size of $*p3$ is not fixed. These structures can be easily represented by our predicates. Particularly, the structure of $*p1$ is represented as $FieldOf(\&(p1 \rightarrow x), p1)$, $FieldOf(\&(p1 \rightarrow y), p1)$, $Long(\&(p1 \rightarrow x))$, and $Long(\&(p1 \rightarrow y))$ (note that the syntax of these predicates is simplified for illustration); $*p2$ is represented as $Array(p2, p2 + 16 * n, 16)$, $FieldOf(\&(p2 \rightarrow x), p2)$ and so on (similar to $*p1$); $*p3$ is represented as $FieldOf(\&(p3 \rightarrow x), p3)$, $Array(\&(p3 \rightarrow z), \&(p3 \rightarrow z) + 16 * m, 16)$, $Long(\&(p3 \rightarrow x))$, and $Long(\&(p3 \rightarrow z))$. \square

Figures 3.11, 3.12, 3.13, and 3.14 presents the probabilistic inference rules. These rules read as follows: the first column is the rule id for easy reference; the second column is the condition that needs to be satisfied in order to introduce the inference rule in the third column. Each inference rule is a first-order logic formula annotated with prior probability.

<u>ID</u>	<u>Condition</u>	<u>Probabilistic Constraint</u>
C_{A01}		$Access(i, v, k) \xrightarrow{p(k)\uparrow} PrimitiveVar(v)$
C_{A02}	$AdjacentChunk(v_1, v_2) \wedge Accessed(v_1) \wedge Accessed(v_2),$	$PrimitiveVar(v_1) \xleftarrow{p\uparrow} PrimitiveVar(v_2)$
C_{A03}	$OverlappingChunk(v_1, v_2) \wedge Accessed(v_1) \wedge Accessed(v_2),$	$PrimitiveVar(v_1) \xleftarrow{p\downarrow} PrimitiveVar(v_2)$
C_{A04}	$Accessed(i, v),$	$PrimitiveVar(v) \xrightarrow{p\uparrow} PrimitiveAccess(i, v)$
C_{A05}	$Accessed(i, v'),$	$PrimitiveAccess(i, v) \xrightarrow{p\uparrow} PrimitiveVar(v')$
C_{A06}	$AccessSingleChunk(i, v.a.r) \wedge Access(i, v, k)$	$PrimitiveAccess(i, v) \xrightarrow{p(k)\uparrow} Scalar(v)$
C_{A07}	$AdjacentChunk(v_1, v_2) \wedge Access(i_1, v_1, k_1) \wedge$ $Access(i_2, v_2, k_2) \wedge AccessSingleChunk(i_1, v_1.a.r) \wedge$ $AccessSingleChunk(i_2, v_2.a.r)$	$Scalar(v_1) \xleftarrow{p(k_1, k_2)\uparrow} Scalar(v_2)$
C_{A08}		$Scalar(v) \xleftarrow{p(k)\downarrow} FieldOf(v, a)$

Figure 3.11. Probabilistic inference for primitive and scalar variables

Each predicate instantiation is associated with a random variable whose posterior probability will be computed by inference. For example, rule C_{A01} means that v has $p(k)$ probability of being a primitive variable if an instruction i has accessed it k times (over all the sample runs). Note that the probability is a function of k . The up-arrow denotes that if $Access(i, v, k)$ is likely, then $PrimitiveVar(v)$ is likely. A down-arrow denotes the opposite.

Primitive Variable and Scalar Variable Recovery. Rules C_{A01} - C_{A05} (In Figure 3.11) are to identify primitive variables. Rule C_{A02} means that a variable is likely primitive if its adjacent one is likely primitive; C_{A03} means that if two variables have overlapping address, one likely being primitive renders the other one unlikely (note the down-arrow); C_{A04} and C_{A05} state that if a variable v is primitive, the instruction that accesses it is a primitive access such that another variable v' accessed by it is primitive too. Rules C_{A06} - C_{A08} are for scalar variable recovery. A primitive variable may not be a scalar variable as it could be a field or an array element. C_{A06} says v is scalar if it is primitive and there is an instruction i that exclusively accesses it. Intuitively, if i accesses (non-scalar) array elements or structure fields, it likely accesses multiple memory chunks. C_{A07} says a scalar's neighbor may be a scalar too, depending on their access frequencies (e.g., when the frequencies are similar). C_{A08} says a scalar variable cannot be a field.

<u>ID</u>	<u>Condition</u>	<u>Probabilistic Constraint</u>
C_{B01}		$MayArray(a, k, s) \xrightarrow{p\uparrow}$ $Array(a, a + s \times k, s) \wedge ArrayStart(a)$
C_{B02}	$AccessMultiChunks(i, r) \wedge$ $LoAddrAccessed(i, r, v_1.a) \wedge$ $HiAddrAccessed(i, r, v_2.a)$	$PrimitiveAccess(i, v_1) \wedge PrimitiveAccess(i, v_2) \xrightarrow{p\uparrow}$ $Array(v_1.a, v_2.a + v_2.s, v_1.s)$
C_{B03}	$(a_{1l} \leq a_{2l} \leq a_{1h} \leq a_{2h}) \wedge$ $(s_1 = s_2 = s) \wedge (s \mid a_{2l} - a_{1l})$	$Array(a_{1l}, a_{1h}, s_1) \xleftrightarrow{p\uparrow} Array(a_{2l}, a_{2h}, s_2)$ $Array(a_{1l}, a_{1h}, s_1) \wedge Array(a_{2l}, a_{2h}, s_2) \xrightarrow{p\uparrow}$ $Array(a_{1l}, a_{2h}, s)$
C_{B04}	$(a_{1l} \leq a_{2l} \leq a_{1h} \leq a_{2h}) \wedge$ $((s_1 \neq s_2) \vee ((s_1 = s_2 = s) \wedge (s \nmid a_{2l} - a_{1l})))$	$Array(a_{1l}, a_{1h}, s_1) \xleftrightarrow{p\downarrow} Array(a_{2l}, a_{2h}, s_2)$ $Array(a_{1l}, a_{1h}, s_1) \xrightarrow{p\uparrow} Array(a_{1l}, a_{2l}, s_1)$ $Array(a_{2l}, a_{2h}, s_2) \xrightarrow{p\uparrow} Array(a_{1h}, a_{2h}, s_2)$
C_{B05}	$a_1 \leq v.a \leq a_2$	$Scalar(v) \xleftrightarrow{p\downarrow} Array(a_1, a_2, s)$ $Array(a_1, a_2, s) \wedge Scalar(v) \xrightarrow{p\uparrow} Array(a_1, v.a, s)$ $Array(a_1, a_2, s) \wedge Scalar(v) \xrightarrow{p\uparrow} Array(v.a + v.s, a_2, s)$
C_{B06}	$a_2 - a_1 < s$	$Array(a_1, a_2, s) = false$
C_{B07}	$BaseAddr(i, v, a) \wedge AccessMultiChunks(i, v.a.r)$	$PrimitiveAccess(i, v) \xrightarrow{p\uparrow} ArrayStart(a)$
C_{B08}	$MostFreqAddrAccessed(i, r, v, k) \wedge$ $AccessMultiChunks(i, r)$	$PrimitiveAccess(i, v) \xrightarrow{p(k)\uparrow} ArrayStart(v.a)$
C_{B09}	$Accessed(i, v_1) \wedge Accessed(i, v_2) \wedge$ $SameRegion(v_1.a, v_2.a) \wedge (v_1.a < v_2.a)$	$ArrayStart(v_1.a) \xrightarrow{p\downarrow} ArrayStart(v_2.a)$

Figure 3.12. Probabilistic inference for arrays

Array Recovery. Rules C_{B01} - C_{B09} (In Figure 3.12) are for array recovery. A common observation is that, the vast majority of arrays are visited in loops. If multiple elements on a continuous region are accessed by an instruction, intuitively, it's likely that this is access to an array. In particular, rules C_{B01} - C_{B02} receive the basic array hints from the previous analysis steps; C_{B03} - C_{B06} aggregate hints to enhance confidence and/or derive new arrays; and C_{B07} - C_{B09} derive array heads. Intuitively, C_{B01} states that there is likely an array if our deterministic reasoning says so (e.g., by observing `calloc`). C_{B02} says if addresses are accessed by the same instruction, there is likely an array and the lowest and highest addresses accessed by the instruction form the lower and upper bounds of an array, respectively. C_{B03}

<u>ID</u>	<u>Condition</u>	<u>Probabilistic Constraint</u>
C_{C01}		$ConstantAllocSize(i, s) \xrightarrow{p\uparrow} UnfoldableHeap(i, s) \wedge FoldableHeap(i, 0)$
C_{C02}		$AllocUnit(i, s) \xrightarrow{p\uparrow} MutitleHeap(i, s)$
C_{C03}	$v.a.r = \mathcal{H}_i$	$PrimitiveVar(v) \xrightarrow{p\uparrow} UnfoldableHeap(i, v.a.o + v.s)$
C_{C04}	$s_1 \neq s_2$	$UnfoldableHeap(i, s_1) \xleftrightarrow{p\downarrow} UnfoldableHeap(i, s_2)$
C_{C05}	$s_1 \leq s_2$	$UnfoldableHeap(i, s_1) \xrightarrow{p\uparrow} UnfoldableHeap(i, s_2)$
C_{C06}	$(a_1.r = a_2.r = \mathcal{H}_i) \wedge (s_1 = s_2)$	$Array(a_1, a_2, s_1) \xrightarrow{p\uparrow} FoldableHeap(i, s_2)$
C_{C07}	$Accessed(v) \wedge (v.a.r = \mathcal{H}_i) \wedge (v.a.o \geq s_h + s_t)$	$PrimitiveVar(v) \wedge UnfoldableHeap(i, s_h) \wedge FoldableHeap(i, s_t) \xrightarrow{p\uparrow}$ $PrimitiveVar(\langle \langle v.a.r, (v.a.o - s_h) \% s_t + s_h \rangle, v.s \rangle)$ $UnfoldableHeap(i, s_h) \wedge FoldableHeap(i, s_t) \xleftrightarrow{p\downarrow} PrimitiveVar(v)$

Figure 3.13. Probabilistic inference for heap folding

says that when two arrays overlap, have the same element size s and the distance of the two arrays is divisible by s , the two arrays can enhance each other's confidence (the first formula) and they can be merged to a larger array (the second formula). C_{B04} says that when two arrays overlap, but they are not homomorphic (e.g. having different element sizes or misalign), one likely being true array renders the other unlikely (the first formula) and the non-overlapping parts can still be considered possible arrays (the second and third formulas). C_{B05} says that a scalar appearing within the range of an array breaks it to two smaller arrays.

Heap Folding. Rules C_{C01} - C_{C07} (In Figure 3.13) are auxiliary rules for analysing heap structures. While BDA can achieve alias analysis accuracy similar to dynamic analysis (with better coverage), it leads to sparse heap behaviors. For example, assume a large heap array of structures is allocated. Different paths may access different heap array elements (at distinct addresses), each disclosing part of the behavior of the structure. Since our goal is to recover the complete structural properties, we need to aggregate these sparse behaviors.

We observe any heap region allocated can be partitioned into two *consecutive* parts: *unfoldable* and *foldable*, while such a region may be a singleton structure with fixed size, an array of structures of a fixed size, or a singleton structure with varying size. The three

allocations at lines 5-7 in Figure 3.10a denote such different cases. *The unfoldable part includes all the fields whose accesses always occur at the same addresses*, whereas *the foldable part includes the fields whose accesses may occur at different (sparse) addresses*. We propose to fold the behaviors of all the instances in the foldable part to the first instance, which will hence possess all the structural properties of all the instances. For example, as shown in Figure 3.10b, the heap region of \mathcal{H}_{05} has only unfoldable fields as $p1 \rightarrow x$ and $p1 \rightarrow y$ always have the addresses of $\langle \mathcal{H}_{05}, 0 \rangle$ and $\langle \mathcal{H}_{05}, 16 \rangle$, respectively. In contrast, all fields in the region \mathcal{H}_{06} are foldable as the $p2[*].x$'s have various addresses. We hence want to fold the behaviors of $p2[1]$, $p2[2]$, and so on to $p2[0]$. The region \mathcal{H}_{07} has an unfoldable field followed by a foldable field which is an array of varying size. *Observe that foldable fields can only occur after unfoldable fields in a region*. In Figure 3.9, we introduce $UnfoldableHeap(i, s)$ to denote the first s bytes of the heap region allocated at i are unfoldable and $FoldableHeap(i, s)$ to denote the region allocated at i has a foldable part with an element size of s . For example, we have $FoldableHeap(7, 16)$ for region \mathcal{H}_7 in Figure 3.10b.

C_{C01} states that if i only allocates a constant size region, the entire region is unfoldable. C_{C02} says that if through deterministic analysis, we know that the allocation size of i is a multiple of s , the foldable part has an element size of s . C_{C03} says that if a primitive field v is found inside a heap region, all the part up to v is unfoldable. This is because unfoldable fields must precede foldable fields. C_{C04} states that a heap region cannot have different unfoldable parts. However, the presence of a smaller unfoldable part can enhance the confidence of a larger unfoldable part (C_{C05}). C_{C06} says that an array found inside a heap region must belong to the foldable part. Rule C_{C07} is the folding rule. The first formula says that a primitive field v found inside a later structure instance inside the foldable region indicates the presence of a primitive field at the corresponding offset inside the first instance. For example in \mathcal{H}_{06} , the identification of y field in $p2[1]$ indicates the presence of y field in $p2[0]$, although $p2[0] \rightarrow y$ is never seen during sample runs. The second formula eliminates the primitive field v after it is folded.

Structure Recovery. Like existing work, we leverage the instruction patterns of load-ing base address to recognize a data structure. However, we model its uncertainty using

<u>ID</u>	<u>Condition</u>	<u>Probabilistic Constraint</u>
C_{D01}		$DataFlowHint(a_1, a_2, s) \xrightarrow{p(s)\uparrow} HomoSegment(a_1, a_2, s)$
C_{D02}		$PointsToHint(a_1, a_2, s) \xrightarrow{p(s)\uparrow} HomoSegment(a_1, a_2, s)$
C_{D03}		$UnifiedAccessPntHint(a_1, a_2, s) \xrightarrow{p(s)\uparrow} HomoSegment(a_1, a_2, s)$
C_{D04}	$(0 < a_2 - a_1 < s_1)$	$HomoSegment(a_1, a_1', s_1) \xleftrightarrow{p\uparrow} HomoSegment(a_2, a_2', s_2)$ $HomoSegment(a_1, a_1', s_1) \wedge HomoSegment(a_2, a_2', s_2) \xrightarrow{p\uparrow}$ $HomoSegment(a_1, a_1', a_2 - a_1 + s_2)$
C_{D05}	$(0 < v_1.a - a_1 = v_2.a - a_2 < s) \wedge (v_1.s \neq v_2.s)$	$PrimitiveVar(v_1) \wedge PrimitiveVar(v_2) \xleftrightarrow{p\downarrow} HomoSegment(a_1, a_2, s)$
C_{D06}	$BaseAddr(v_1, i, v_2.a) \wedge Accessed(v_1) \wedge Accessed(v_2)$	$PrimitiveVar(v_1) \wedge PrimitiveVar(v_2) \xrightarrow{p\uparrow} FieldOf(v_1, v_2.a)$
C_{D07}	$v.a.r = \mathcal{H}_i$	$PrimitiveVar(v) \xrightarrow{p\uparrow} FieldOf(v, \langle \mathcal{H}_i, 0 \rangle)$
C_{D08}	$(n \leq s) \wedge (v_1.a = a_1 + n) \wedge (v_2.a = a_2 + n)$	$FieldOf(v_1, a_1) \wedge HomoSegment(a_1, a_2, s) \xrightarrow{p\uparrow} FieldOf(v_2, a_2)$
C_{D10}	$a_1 \neq a_2$	$FieldOf(v, a_1) \xleftrightarrow{p\downarrow} FieldOf(v, a_2)$
C_{D11}	$PointsTo(v_1, v_2.a) \wedge Accessed(v_1) \wedge Accessed(v_2)$	$PrimitiveVar(v_1) \wedge PrimitiveVar(v_2) \xrightarrow{p\uparrow} Pointer(v_1, v_2.a)$

Figure 3.14. Probabilistic inference for structures

probabilities. In addition, we consider the data flow among different variables of the same type. Specifically, rules C_{D01} - C_{D10} are for structure recovery, including global/stack/heap structures. Intuitively, we first identify memory segments (i.e., part of a structure) that are homomorphic, meaning that they have highly similar access patterns, data flow, and points-to relations. These segments are then intersected, unioned, or separated to form the final structures. Individual fields can be then identified from their access pattern within the structure. Specifically, rules C_{D01} - C_{D03} (In Figure 3.14) receive deterministic hints. C_{D04} states that if a pair of homomorphic segments overlap with another pair of homomorphic segments, they enhance each other's confidence (the first formula) and may form a pair of new homomorphic segments that are the union of the original two pairs (the second formula). Intuitively, it corresponds to that the sub-parts of a same complex structure are being exposed differently (e.g., through different data flow), and we leverage the overlap of these parts to join them. C_{D05} says that if the corresponding primitive fields in a pair of homo-

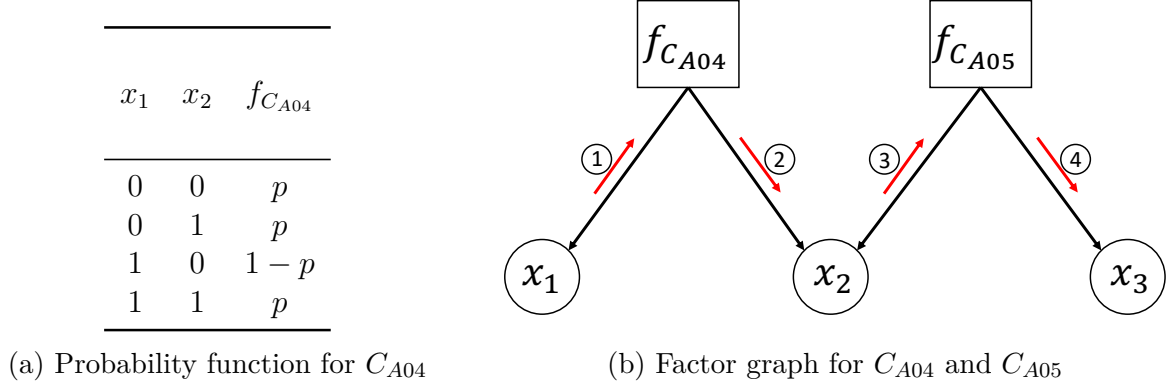


Figure 3.15. Factor graph example

morphic segments have different access patterns (4-byte access versus 8-byte access), either the primitive field predicates are likely false or the homomorphic predicate. Rules C_{D06} and C_{D07} identify fields of structure from the deterministic reasoning results (e.g., *BaseAddr*) and if the accesses are primitive. C_{D08} transfers field information across a pair of homomorphic segments. Rule C_{D09} asserts a field cannot have two different base addresses. C_{D09} determines a pointer variable v_1 if a valid address $v_2.a$ is stored to v_1 and v_2 has been accessed as a primitive variable.

OSPREY also has a set of typing rules that associate primitive types (e.g., int, long, and string) to variables, based on their data-flow to program points that disclose types such as invocations to string library functions. These rules are similar to existing works [33, 43, 44] and hence elided.

3.5.2 Probabilistic Constraint Solving

Each of the probabilistic constraints in Figures 3.11, 3.12, 3.13, and 3.14 (the formulas in the last column) essentially denotes a probability function over the random variables involved. The functions can be further transformed to a probabilistic graph model called *factor graph* [65], which is a bi-partite graph with two kinds of nodes, *function node* denoting a probability function, and *variable node* denoting a random variable. Edges are introduced

$$\begin{aligned}
\textcircled{1}: \quad m_{x_1 \rightarrow f_{C_{A04}}}(x_1) &= 1 \\
\textcircled{2}: \quad m_{f_{C_{A04}} \rightarrow x_2}(x_2 = 0) &= \frac{\sum_{x_1} f_{C_{A04}}(x_1, 0) * m_{x_1 \rightarrow f_{C_{A04}}}(x_1)}{\sum_{x_1, x_2} f_{C_{A04}}(x_1, x_2) * m_{x_1 \rightarrow f_{C_{A04}}}(x_1)} \\
&= \frac{0.8 + 0.2}{0.8 + 0.8 + 0.2 + 0.8} * 1 = \frac{1}{2.6} \\
m_{f_{C_{A04}} \rightarrow x_2}(x_2 = 1) &= \frac{\sum_{x_1} f_{C_{A04}}(x_1, 1) * m_{x_1 \rightarrow f_{C_{A04}}}(x_1)}{\sum_{x_1, x_2} f_{C_{A04}}(x_1, x_2) * m_{x_1 \rightarrow f_{C_{A04}}}(x_1)} \\
&= \frac{0.8 + 0.8}{0.8 + 0.8 + 0.2 + 0.8} * 1 = \frac{1.6}{2.6} \\
\textcircled{3}: \quad m_{x_2 \rightarrow f_{C_{A05}}}(x_2) &= m_{f_{C_{A04}} \rightarrow x_2}(x_2) \\
&\quad \sum_{x_2} f_{C_{A05}}(x_2, 1) * m_{x_2 \rightarrow f_{C_{A05}}}(x_2) \\
\textcircled{4}: \quad m_{f_{C_{A05}} \rightarrow x_3}(x_3 = 1) &= \frac{\sum_{x_2} f_{C_{A05}}(x_2, x_3) * m_{x_2 \rightarrow f_{C_{A05}}}(x_2)}{\sum_{x_2, x_3} f_{C_{A05}}(x_2, x_3) * m_{x_2 \rightarrow f_{C_{A05}}}(x_2)} \\
&= 0.65
\end{aligned}$$

Figure 3.16. Definition and computation of each message shown in Figure 3.15

between a function node and all the variable nodes related to the function. The whole factor graph denotes the joint distribution of all the random variables.

Example. Boolean variables x_1, x_2 , and x_3 denote $PrimitiveVar(v)$, $PrimitiveAccess(i, v)$, and $PrimitiveVar(v')$, respectively. Rules C_{A04} is transformed to $x_1 \xrightarrow{p^\uparrow} x_2$, which denotes the probability function in Figure 3.15a. The probability function for C_{A05} is similar. The two form a factor graph in Figure 3.15b, which could be solved by belief propagation algorithms with passing messages on it. For example, assume the prior probabilities of C_{A04} and C_{A05} are both 0.8, and we want to compute the marginal probability $p(x_3 = 1)$, that is, the probability of v being of primitive type. As the factor graph is a tree, we can call x_3 the root node. Then message passing starts from the leaf node x_1 . After messages reach the root finally, the marginal probability of x_3 can be computed. The definition and computation of each message is shown in Figure 3.16. \square

Given a set of observations (e.g., $x_1 = 1$) from the deterministic reasoning step, and the prior probabilities (p values), posterior marginal probabilities are computed by propagating and updating probabilities along the edges. Some of the rules, such as C_{B02} , generate new

predicate nodes during inference. After each round of inference (i.e., probabilities converge after continuous updates), it checks all the (new) predicate nodes to coalesce those denoting the same meaning to one node. The node inherits all the edges of all the other nodes that are coalesced. Then another round of inference starts. Note that while some probabilistic inference applications are stochastic, our application (variable recovery and typing) has the uncertainty originating from loss of debugging information. In other words, there is deterministic ground truth (or, the ground-truth variables and their types are deterministic). In this context, the number of hints that we can aggregate plays a more important role than the prior probabilities. Graph models provide a systematic way of aggregating these hints, while respecting the inherent structural properties (e.g., control-flow and data-flow constraints). We hence adopt simple prior probabilities, $p \uparrow = 0.8$, $p \downarrow = 0.2$, and $p(k)$ is computed from the ratio between k and the total number of sampled paths in BDA. In fact, there are a number of existing work [59, 66–68] leveraging probabilistic inference for similar applications with (mostly) deterministic ground truth (e.g., specification inference for explicit information flow). They use preset prior probabilities and their results are not sensitive to prior probability configurations. We follow a similar setting.

3.6 Evaluation

To assess the effectiveness of OSPREY, we perform two sets of experiments, using the benchmarks from TIE [43] and Howard [44]. The first set is performed on Coreutils [69], a standard benchmark widely used in binary analysis projects [17, 42–44, 60], consisting of 101 programs. We compare OSPREY with other state-of-the-art binary analysis tools, including Ghidra (version 9.2), Angr (version 8.20) and IDA Pro (version 7.2). We cannot compare with TIE as the system is not available. And we confirmed with the BAP [17] team that BAP does not have TIE as part of it. Another set is performed on the benchmark provided by the Howard project [44], consisting of 5 programs. The purpose is to have a side-by-side comparison with Howard. Since we are not able to acquire the Howard system, the only way to compare is to use their published results and hence the same set of programs. All experiments were conducted on a server equipped with 32-cores CPU (Intel®

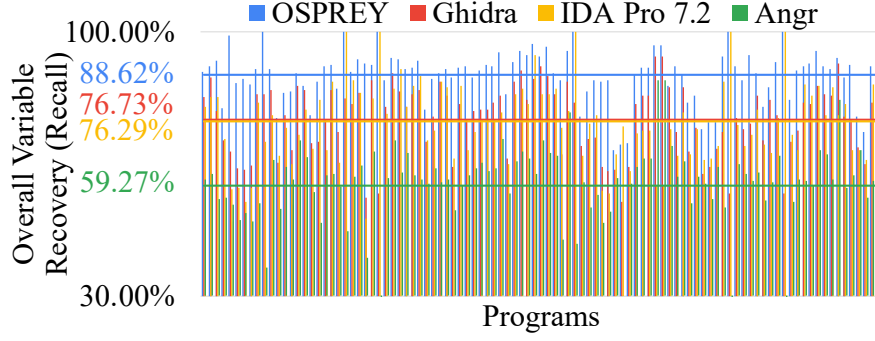


Figure 3.17. Recall for all variables (primitive and complex)

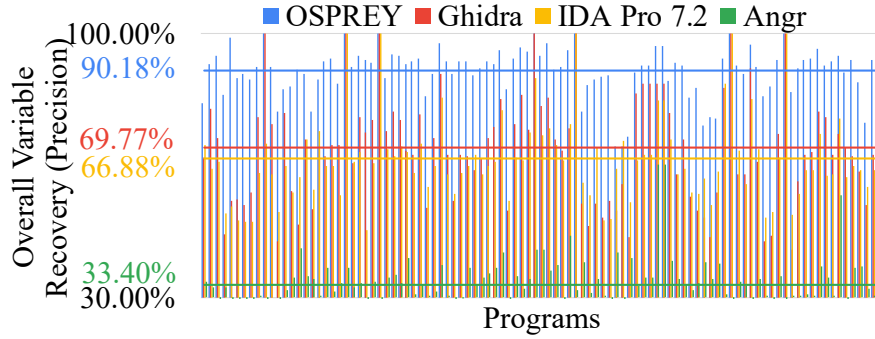


Figure 3.18. Precision for all variables (primitive and complex)

XeonTM E5-2690 @ 2.90GHz) and 128G main memory. To follow a similar setup in TIE and Howard, we use GCC 4.4 to compile the programs into two versions: a version with debugging information used as the ground truth and a stripped version used for evaluation. Our assumption of proper disassembly is guaranteed because GCC does not interleave code and data on Linux [70].

3.6.1 Evaluation on Coreutils

Similar to the standard in the literature [43, 44], we inspect the recovered types (including structure types) of individual variables. If it is a pointer type, we inspect the structure that is being pointed to. For example, if a `(Socket *)` variable is recovered as `(void*)`, we consider it incorrect. We say it is correct only if the variable is recovered as a pointer pointing to a structure homomorphic to `Socket`. The overall recall and precision are shown

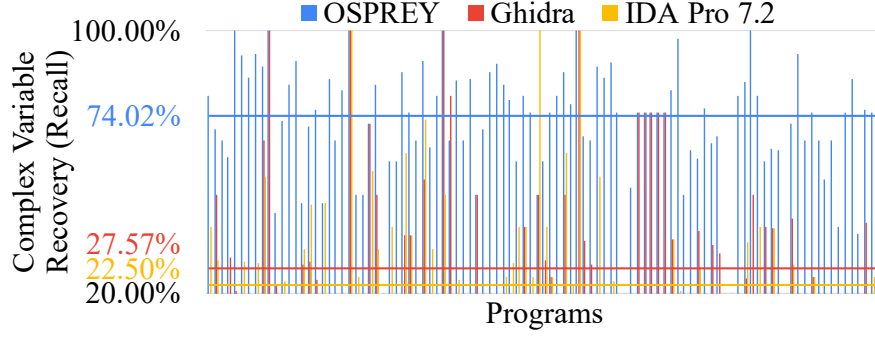


Figure 3.19. Recall for complex variables

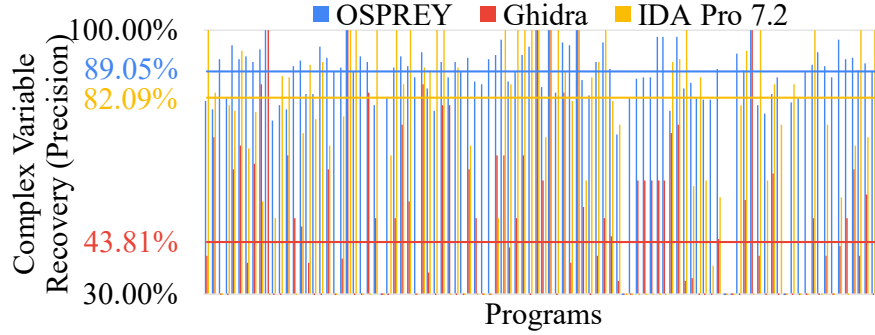


Figure 3.20. Precision for complex variables

in Figure 3.17 and Figure 3.18, respectively. As we can see, OSPREY achieves more than 88% recall, and more than 90% precision, outperforming the best of other tools (i.e., Ghidra with around 77% recall and 70% precision). Figures 3.19 and 3.20 present the recall and precision of complex types recovery. Complex types include structures, unions, arrays and pointers to structures, unions and arrays. Note that Angr could not recover complex data types, hence we do not list its results on the figures. Observe that the recall of OSPREY is around 74%, more than 2 times higher than Ghidra and IDA Pro. The precision of OSPREY also outperforms Ghidra and IDA Pro. One may mention that IDA Pro has a comparable precision rate with OSPREY. The reason is that IDA Pro performs a very conservative type analysis to ensure high precision, leading to a low recall (less than 23%).

Cases Where Ghidra and IDA Pro Do Better. There are few cases where Ghidra and IDA Pro achieve better performance. Further inspection reveals that those are very simple


```

unsigned long sha256(char *msg) {
    struct SHA256 ctx; char *c = msg;
    ...
    while (c) {
        ctx.S0 =
            calculate0(ctx.S0, ctx.S1, c);
        ctx.S1 =
            calculate0(ctx.S0, ctx.S1, c);
        c = get_next_chunk(c);
    }
    return fini(ctx.S0, ctx.S1);
}

```

Figure 3.21. Missing data structures

```

struct my_chunk {
    char buf[0x80];
    struct my_chunk *next;
}

struct my_chunk *xmalloc() {
    struct my_chunk *cur = HEAD;
    while (cur->next & 1)
        cur = (cur->next ^ 1);
    cur->next ^= 1;
    return p;
}

```

Figure 3.22. Misidentified data structures

programs without complex structures (e.g., `struct` or in-stack array), where no conflict will occur during deterministic reasoning. Hence, approaches like Ghidra and IDA Pro can handle them well. *OSPREY* also works well, but may misidentify very few variables due to the infeasible paths produced by BDA.

Missing Data Structures. We find that missing data structures are mainly due to stack-nested `structs` that are never used outside their stack frames. Consider the code snippet from *sha256sum* in Figure 3.21, where a stack-nested structure `SHA256 ctx` is allocated on stack and used exclusively within the function. As such, *OSPREY* cannot gather any valuable hints about `ctx`. That is also the major reason that *OSPREY* has relatively large tree difference for those hashing binaries (e.g., *sha256sum*) in Figure 3.23 in Appendix.

Misidentified Data Structures. In our benchmarks, custom heap allocators are a major source of misidentified data structures by *OSPREY*. Consider a simplified `xmalloc` from *grep* in Figure 3.22. Its basic allocation unit is called `my_chunk`, consisting of a buffer `buf` and a pointer `next`. Different from common pointers, `my_chunk.next` uses its last bit to indicate whether this chunk is in use (in normal case, the last bit is always zero due to memory alignment). Thus, at line 5, `xmalloc` finds the first chunk whose in-use bit is not set, sets the bit, and returns the chunk. As a result, `my_struct.next` can point to a `struct`

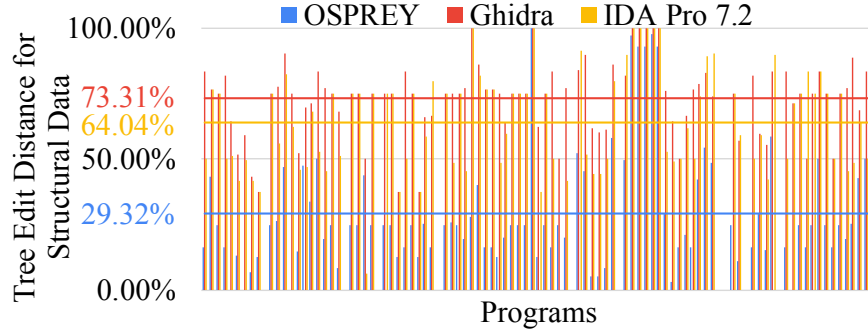


Figure 3.23. Tree difference for CoreUtils

`my_struct` or `char my_struct.buf[1]` (both are common cases). These confusing *PointsTo* hints misled OSPREY to falsely recover unions. Other reasons include insufficient hints.

To better quantify our results on complex variables, we construct a syntax tree for each complex type (with fields being the child nodes). Nesting structures and unions are precisely modeled, and any inner nesting structure or union type without outer references are ignored. Cycles are removed using a leaf node with a special type tag. We then compare the edit distance of the recovered trees and the ground-truth trees. We compute *tree difference* that is defined as the ratio of the tree edit distance (i.e., the minimum number of tree edits that transform a tree into another) and the whole tree size. The smaller the tree difference, the better the recovery result. Figure 3.23 in Appendix shows the results. Overall, OSPREY has the minimal tree difference, which is 2.50 and 2.18 times smaller than Ghidra and IDA Pro.

3.6.2 Evaluation on *Howard Benchmark*

The results for the Howard benchmark are shown in Table 3.1. OSPREY substantially outperforms Ghidra, IDA Pro and Angr, especially for complex variables, in all metrics (recall, precision and tree difference) For all variables, the precision improvement over Ghidra, IDA Pro, and Angr is 30.64%, 41.11%, and 67.77%, respectively, and the recall improvement is 24.98%, 36.78%, and 50.49%, respectively. For complex variables, the precision improvement over Ghidra and IDA Pro is 42.64% and 27.09%, respectively, and the recall improvement is 53.78% and 65.36%, respectively. Our tree differences are 6.31 and 3.20 times smaller than

Table 3.1. Analysis results of Howard benchmark

Metric	Program	Osprey		Ghidra		IDA Pro		Angr	
		Reca.	Prec.	Reca.	Prec.	Reca.	Prec.	Reca.	Prec.
<i>Overall Variable</i>	wget	85.32	86.14	66.83	62.94	62.82	60.02	39.94	26.96
	lighttpd	97.67	97.67	52.65	52.15	46.18	41.37	44.35	22.90
	grep	82.10	84.07	67.63	69.34	67.09	63.97	46.64	30.53
	gzip	100.0	100.0	84.78	79.10	78.26	75.00	59.78	37.42
	fortune	100.0	100.0	68.29	51.16	26.83	22.00	21.95	11.25
	Avg.	93.02	93.58	68.04	62.94	56.24	52.47	42.53	25.81
<i>Complex Variable</i>	wget	73.26	83.14	29.21	47.20	20.29	76.39	00.00	N/A
	lighttpd	100.0	95.44	05.51	27.08	06.78	50.00	00.00	N/A
	grep	57.39	84.52	10.43	35.29	11.30	41.67	00.00	N/A
	gzip	100.0	100.0	66.67	73.68	57.14	81.25	00.00	N/A
	fortune	100.0	100.0	50.00	66.67	00.83	33.33	00.00	N/A
	Avg.	86.13	92.62	32.35	49.98	20.77	65.53	00.00	N/A
<i>Tree Difference</i>	wget	28.92		70.99		62.84		100.0	
	lighttpd	00.00		80.18		64.87		100.0	
	grep	30.09		78.41		60.93		100.0	
	gzip	00.00		42.50		00.00		100.0	
	fortune	00.00		100.0		00.00		100.0	
	Avg.	11.80		74.42		37.73		100.0	

Ghidra and IDA Pro. Compared to Coreutil programs, these programs are more complex, providing more hints to OSPREY. Especially in the complex variable recovery for lighttpd, OSPREY has 100% recall and 95% precision, while Ghidra has 5.5% recall and 27% precision, IDA Pro 6.8% and 50%. Manual inspection discloses that lighthttp has a large number of structures on heap, providing ample hints for OSPREY.

We also perform side-by-side comparison with Howard. Since Howard is not available (after communicating with its authors), we use the data reported in the paper. We use the same setup and report results in the same metrics. Table 3.2 presents the coverage of functions (columns 2~3) and variables (column 4~5), and the accuracy of type discovery for stack variables measured in the number of variables (columns 6~7) and in the number of bytes (columns 8~9). OSPREY outperforms, especially for coverage. We cannot com-

Table 3.2. Comparison between OSPREY and Howard

Program	Cov of Funcs (%)		Cov of Vars (%)		Accuracy (% of vars)		Accuracy (% of bytes)	
	OSPREY	Howard	OSPREY	Howard	OSPREY	Howard	OSPREY	Howard
wget	100%	51%	100%	56%	87.28%	77.03%	71.58%	63.12%
lighttpd	100%	55%	100%	62%	97.89%	86.15%	95.30%	82.10%
grep	100%	50%	100%	56%	100.00%	86.15%	100.00%	87.10%
gzip	100%	70%	100%	81%	86.17%	75.07%	79.13%	78.04%
fortune	100%	71%	100%	77%	100.00%	83.04%	100.00%	82.10%
Avg.	100%	59%	100%	66%	94.27%	81.49%	89.20%	78.50%

Table 3.3. Average F_1 scores for OSPREY with different prior probabilities

	$p \uparrow = 0.7$	$p \uparrow = 0.8$	$p \uparrow = 0.9$
$p \downarrow = 0.1$	0.915	0.929	0.923
$p \downarrow = 0.2$	0.931	0.933	0.923
$p \downarrow = 0.3$	0.919	0.930	0.924

pare with their heap results as Howard measured the accuracy for individual dynamic heap objects.

3.6.3 Sensitivity Analysis

We analyze the sensitivity of OSPREY’s accuracy on the prior probabilities $p \uparrow$ and $p \downarrow$. Table 3.3 shows the average F_1 scores [71] for the programs in the Howard benchmark set, with $p \uparrow$ varying from 0.7 to 0.9 and $p \downarrow$ from 0.1 to 0.3. We elide other metrics as they reveal similar trendings. Note that the F_1 scores vary within a limited range, less than 2%, with different prior probabilities. It supports that OSPREY is robust against the prior probability changes.

3.6.4 Execution Time

In Table 3.4, we measure the execution time of different tools on the two benchmark sets. In general, OSPREY is 18.57, 88.04, and 50.79 times slower than Ghidra, IDA Pro, and Angr, respectively. We argue that reverse engineering is often a one-time effort and OSPREY provides a different trade-off between cost and accuracy. It is also worth noting that Ghidra

Table 3.4. Execution time of different tools. The numbers in the brackets denote how many times OSPREY is slower than the corresponding tool.

Program	Osprey	Ghidra		IDA		Angr	
<i>Howard</i>	wget	3604.80s	94.74s (37.05×)	18.98s (188.88×)	41.47s (85.92×)		
	lighttpd	2013.12s	63.89s (30.51×)	16.80s (118.83×)	31.60s (62.70×)		
	grep	832.52s	66.75s (11.47×)	32.62s (24.52×)	33.88s (23.57×)		
	gzip	483.65s	52.84s (8.15×)	11.84s (39.84×)	18.57s (25.04×)		
	fortune	422.92s	37.48s (10.28×)	6.30s (66.13×)	7.11s (58.45×)		
<i>CoreUtils</i>		528.24s	35.35s (13.94×)	5.80s (90.08×)	10.55s (49.07×)		
Avg.	1314.21s	58.51s	(18.57×)	15.39s	(88.04×)	23.87s	(50.79×)

is the second slowest one due to its register-based data-flow analysis, and IDA Pro is the fastest one as its variable recovery mainly relies on hard-coded code pattern matching rules.

3.6.5 Scalability

To assess the scalability of OSPREY, we evaluate OSPREY on Apache and Nginx, two well-known applications with significantly larger code base than the benchmarks we used. The results are shown in Figure 3.24. On both programs, OSPREY produces the highest F_1 Score for overall and complex variable recovery, and the lowest tree difference. Although OSPREY takes around an hour and a half, we argue that it is a reasonable overhead for binary analysis and reverse engineering.

3.6.6 Impact of Aggressive Optimization

To understand the impact of aggressive optimizations, we evaluate OSPREY on the two benchmark sets compiled with -O3, the most aggressive builtin optimization flag of GCC.

The results are shown in Table 3.5. We calculate the F_1 score [71] for each tool, and summarize CoreUtils’ results. Table 3.5a presents the overall F_1 scores including both scalar and complex variables. The average F_1 scores (with -O3) for OSPREY, Ghidra, IDA Pro, and Angr are 0.70, 0.48, 0.27, and 0.16, respectively; and the degradation from the default optimization (-O0) are 23.84%, 27.18%, 45.47%, and 42.64%, respectively. Although recovering

Table 3.5. Impact of aggressive optimizations with -O3. **Def.**, **O3**, **Degra.**, and **# CVars** denote the analysis results for binaries compiled under the default optimization (-O0), under -O3, degradation from -O0, and the number of complex variables in memory, respectively.

(a) F_1 scores for overall variable recovery

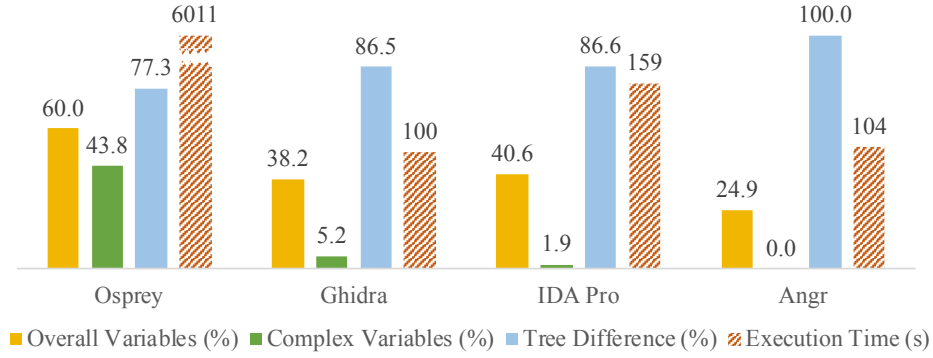
Program		Osprey			Ghidra			IDA			Angr		
		Def.	O3	Degra.	Def.	O3	Degra.	Def.	O3	Degra.	Def.	O3	Degra.
Howards	wget	0.86	0.66	23.11%	0.65	0.51	20.89%	0.48	0.20	58.65%	0.32	0.21	34.52%
	lighttpd	0.98	0.65	33.42%	0.52	0.29	44.59%	0.43	0.15	64.07%	0.30	0.09	71.72%
	grep	0.83	0.74	11.21%	0.68	0.60	11.74%	0.54	0.20	63.03%	0.37	0.16	55.72%
	gzip	1.00	0.74	26.44%	0.82	0.37	55.20%	0.67	0.24	64.30%	0.46	0.16	64.46%
	fortune	1.00	0.82	17.86%	0.58	0.63	-7.16%	0.24	0.33	-38.36%	0.15	0.20	-34.44%
CoreUtils		0.89	0.62	31.01%	0.74	0.49	37.78%	0.71	0.27	61.12%	0.43	0.16	63.84%
Avg.		0.93	0.70	23.84%	0.67	0.48	27.18%	0.51	0.23	45.47%	0.34	0.16	42.64%

(b) F_1 scores for complex variable recovery

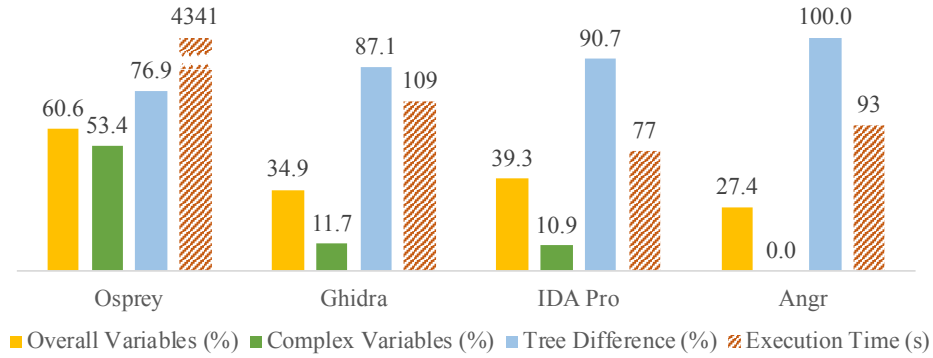
Program		Osprey		Ghidra		IDA		# CVars	
		Def.	O3	Def.	O3	Def.	O3	Def.	O3
Howards	wget	0.78	0.55	0.36	0.45	0.32	0.14	239	127
	lighttpd	0.98	0.44	0.09	0.38	0.12	0.33	318	43
	grep	0.68	0.50	0.16	0.35	0.18	0.20	120	38
	gzip	1.00	0.55	0.70	0.35	0.67	0.33	45	41
	fortune	1.00	0.76	0.57	0.71	0.13	0.52	16	13
CoreUtils		0.80	0.62	0.38	0.43	0.39	0.35	23	11
Avg.		0.87	0.57	0.38	0.45	0.30	0.31	127	45

(c) Tree difference

Program		Osprey			Ghidra			IDA		
		Def.	O3	Degra.	Def.	O3	Degra.	Def.	O3	Degra.
Howards	wget	28.92	57.24	49.47%	70.99	72.88	02.48%	62.84	75.14	16.37%
	lighttpd	00.00	21.42	100.0%	80.18	55.10	-45.53%	64.87	62.81	-03.28%
	grep	30.09	26.96	-11.63%	78.41	72.62	-07.97%	60.93	89.68	32.06%
	gzip	00.00	41.67	100.0%	42.50	62.50	32.00%	00.00	50.00	100.0%
	fortune	00.00	08.00	100.0%	100.0	50.00	-100.0%	00.00	50.00	100.0%
CoreUtils		29.32	63.26	53.65%	73.31	78.69	06.83%	64.04	78.61	18.54%
Avg.		14.72	36.42	65.25%	74.23	65.28	-18.70%	42.11	67.71	43.95%



(a) Apache



(b) Nginx

Figure 3.24. Analysis results for Apache and Nginx

accurate types from aggressively optimized code is very challenging, OSPREY substantially outperforms other state-of-the-art techniques. Besides, OSPREY is the most robust tool among all the evaluated ones. Manual inspection discloses that some aggressive optimizations disrupt OSPREY’s hints (e.g., loop unrolling [72] and partial function inlining [73]), resulting in the degraded accuracy. For example, loop unrolling can generate multiple copies of a single memory access instruction such that we lose the hint that detects an array by observing consecutive memory locations being accessed by the same instruction.

Table 3.5b shows the F_1 scores for complex variable recovery. Observe that OSPREY still achieves substantially better F_1 of 0.57 (compared to 0.45 for Ghidra and 0.31 for IDA Pro). One may notice that Ghidra and IDA Pro get better results with the -O3 flag. Although it seems counter-intuitive, further inspection shows that it is not because they are having better performance but rather the number of complex variables in memory becomes smaller. Recall

that we consider a structure being pointed to by a pointer in memory a complex variable. With -O3, these pointers are largely allocated to registers. We do not collect results for these cases as Howard does not consider variables in registers. While Ghidra and IDA Pro tend to have trouble with complex variables in memory, the number of such cases are reduced.

We additionally count the number of complex variables, shown in Table 3.5b. The results show that the number of complex variables decreases a lot from the default setting (127 v/s 45), supporting our hypothesis.

Table 3.5c presents the tree difference. Although OSPREY has the smallest tree difference of 36.42 (compared to 65.28 for Ghidra and 67.71 for IDA Pro), the aggressive optimizations have larger impact on OSPREY. This is however reasonable because OSPREY’s structure recovery mainly depends on hints from program behaviors which can be greatly changed by optimizations, while Ghidra and IDA Pro mainly depend on predefined function prototypes of external library calls which are rarely influenced by optimizations. Ghidra’s register-based data-flow analysis also benefits from optimizations. We foresee that a set of rules particularly designed for optimized programs can be developed for OSPREY. We will leave that to our future work.

Finally, we want to point out that *fortune* is an outlier which always achieves better results under aggressive optimizations. This is because *fortune* is a very simple program (randomly outputting predefined sentences [74]) and O3 optimizations put most of its variables in registers, reducing aliasing and greatly benefiting the register-based data-flow analysis.

3.6.7 Impact of Different Compilers

To study the robustness over different compilers, we additionally examine OSPREY on benchmarks compiled by Clang [75], another mainstream compiler. We use Clang 6.0 to compile the two benchmark sets with the default and -O3 optimization flags, and summarize the results in Figure 3.25. The results show that OSPREY has good robustness with different compilers under the default compilation setting (less than 6% difference for each program). Although there is a larger difference between GCC and Clang under the -O3 setting, we speculate that it is because the -O3 optimizations of GCC and Clang behave differently

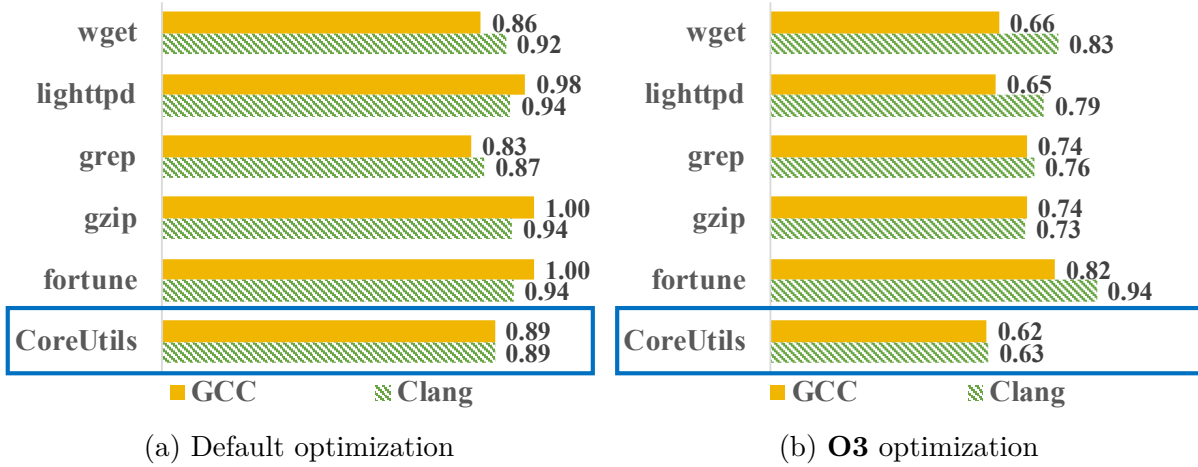


Figure 3.25. OSPREY’s F_1 scores for overall variable recovery on the two benchmark sets compiled by GCC and Clang. The results of CoreUtils are averaged over all programs.

Table 3.6. Effects of BDA and probabilistic inference. **Original**, **w/o BDA**, and **w/o Prob.** stand for the original OSPREY, OSPREY with a dynamic-execution component instead of BDA, and OSPREY with deterministic inference instead of probabilistic inference, respectively. **Cov.** denotes the fraction of functions that the dynamic approach exercised.

Program	Original		w/o BDA			w/o Prob.	
	Recall	Precision	Recall	Precision	Coverage	Recall	Precall
wget	85.32	86.14	29.46	86.31	51%	45.43	47.21
lighttpd	97.67	97.67	73.75	97.16	55%	40.24	40.74
grep	82.10	84.07	44.48	89.78	50%	44.76	46.04
gzip	100.0	100.0	43.48	100.0	74%	64.37	64.37
fortune	100.0	100.0	75.61	100.0	76%	78.57	78.57
Avg.	93.02	93.58	53.36	94.65	61%	54.67	55.39

(e.g., they have different thresholds for loop unrolling). The results of complex variable recovery and tree difference reveal similar trends and are hence elided.

3.6.8 Contribution Breakdown of Different Components

To better understand the effect of different components, including BDA and probabilistic inference, we further evaluate OSPREY with two variations. Specifically, to study the con-

tributions of BDA, in the first variation, we replace the BDA component with a dynamic-execution component built upon Pintools [76]. Following the same setup as Howard, we use the provided test suite and also KLEE to increase code coverage. To study the effect of probabilistic inference, in the second variation, we turn the probabilistic inference to deterministic inference. The deterministic inference rules are largely derived from the probabilistic rules but have the probabilities removed. As such, when multiple contradictory inference results are encountered (e.g., conflicting types for a variable), which are inevitable due to the inherent uncertainty, the algorithm randomly picks one to proceed.

The results are shown in Table 3.6. We report the precision and recall of the first variation for overall variables in the fourth and fifth columns. We also report the dynamic code coverage in the sixth column. Due to page limits, we elide other metrics as they are less interesting. Compared with the original OSPREY, the dynamic-execution-based OSPREY has slightly higher precision but lower recall. As dynamic execution strictly follows feasible paths, there are fewer conflicts, benefiting the precision. However, the conflicts introduced by BDA’s incapacibilities of determining infeasible paths are decentralized and cumulatively resolved by the large number of hints, making the improvement limited. On the other hand, the dynamic-execution-based OSPREY cannot get hints from the non-executed functions, leading to the low recall. Hence, we argue that BDA is essential to OSPREY.

The results of the second variation are shown in the last two columns of Table 3.6. Note that the deterministic version of OSPREY has nearly 40% decrease in terms of both recall and precision. Such results indicate the probabilistic parts of OSPREY are critical. We also study the reason behind the degradation. On one hand, due to the infeasible paths, BDA may generate many invalid accesses. When these accesses conflict with the valid ones, the deterministic algorithm may choose the wrong one. On the other hand, many inference rules / hints have inherent uncertainty. For example, rule C_{B02} says when an instruction accesses multiple addresses in the same region, likely, there is an array in that region. Note that it is likely but not certain, as the situation could also be that a pointer points to multiple individual objects. Deterministic approaches are by their nature not suitable for handling such inherent uncertainty.

```

1 int network_rxxx(server *srv)
2 {
3     server *v1; // rbx
4     int result; // rax
5     size_t v3; // rbp
6     server_socket *v4; // r12
7     fdnode *v5; // rax
8     fdevents *v6; // rdi
9
10    v1 = srv;
11    result = fdevent_sxxx(srv->ev);
12    if ( result != -1 )
13    {
14        v3 = 0LL;
15        if ( !srv->sockets_disabled )
16        {
17            while ( v1->srv_sockets.
18                    used > v3 )
19            {
20                v4 = v1->srv_sockets.
21                    ptr[v3++];
22                v5 = fdevent_gxxx(
23                    v1->ev,
24                    v4->fd,
25                    network_sxxx, v4
26                );
27                v6 = v1->ev;
28                v4->fdn = v5;
29                fdevent_fxv(v6, v5, 1);
30            }
31        }
32        result = 0LL;
33    }
34    return result;
35 }

```

(a) Ground truth

```

1 __int32 __fastcall sub_D840(__int64 a1)
2 {
3     __QWORD *v1; // rbx
4     __int32 result; // rax
5     unsigned __int64 v3; // rbp
6     __int64 v4; // r12
7     __int64 v5; // rax
8     __int64 v6; // rdi
9
10    v1 = (__QWORD *)a1;
11    result = sub_12B7A((__QWORD *)a1 + 24));
12    if ( (_DWORD)result != -1 )
13    {
14        v3 = 0LL;
15        if ( !*(__DWORD *)a1 + 100 )
16        {
17            while ( v1[2] > v3 )
18            {
19                v4 = (__QWORD *)(*v1 + 8 * v3++);
20                v5 = sub_21860(
21                    v1[3],
22                    *(unsigned int *)v4 + 112,
23                    sub_18F30, v4
24                );
25                v6 = v1[3];
26                *(__QWORD *)v4 + 120 = v5;
27                sub_219C0(v6, v5, 1);
28            }
29        }
30        result = 0;
31    }
32    return result;
33 }

```

(b) Vanilla IDA Pro 7.2

```

1 __int32 __fastcall sub_D840(struct_C264 *a1)
2 {
3     struct_C264 *v1; // rbx
4     __int32 result; // rax
5     unsigned __int64 v3; // rbp
6     struct_CF4A *v4; // r12
7     struct_12A42 *v5; // rax
8     struct_12A0E *v6; // rdi
9
10    v1 = a1;
11    result = sub_12B7A(a1->ptr_field_28);
12    if ( result != -1 )
13    {
14        v3 = 0LL;
15        if ( !a1->dat_field_74 )
16        {
17            while ( v1->dat_field_10 > v3 )
18            {
19                v4 = v1->ptr_ptr_field_0[v3++];
20                v5 = sub_21860(
21                    v1->ptr_field_28,
22                    v4->dat_field_10,
23                    sub_18F30, v4
24                );
25                v6 = v1->ptr_field_28;
26                v4->ptr_field_18 = v5;
27                sub_219C0(v6, v5, 1);
28            }
29        }
30        result = 0;
31    }
32    return result;
33 }

```

(c) IDA Pro 7.2 w/ OSPREY

Figure 3.26. Decompiled results for `lighttpd`'s function `network_register_fdevents`

3.7 Applications

3.7.1 Improving IDA Decompilation

Decompilation transforms low level binary code to human-readable high-level program. The readability of decompiled code hinges on the recovery of variables and data structures. To investigate how OSPREY improves decompilation in IDA, we implement an IDA plugin to feed the decompiler of IDA with the recovered information provided by OSPREY. In Figure 3.26 and 3.27, we show a case study on the decompilation of `lighttpd`'s function `network_register_fdevents`. The ground truth, the decompilation results of the vanilla IDA, and of the enhanced IDA are presented in the three columns, respectively. IDA can precisely recover some primitive variables (e.g., `result` at line 4 and `v3` at line 5), but fails to recover the complex data structures (e.g., `v4` at line 6, which is a pointer to a `server_socket`

<pre> struct server { struct server_socket_array { struct server_socket { sockaddr addr; int fd; unsigned short is_ssl; unsigned short sidx; fdnode *fdn; buffer *srv_token; } **ptr; size_t size; size_t used; } srv_sockets; fdevents *ev; ... int sockets_disabled; ... } </pre>	<pre> struct struct_C264 { struct struct_CF4A { sockaddr dat_field_0; __int32 dat_field_10; unsigned __int16 field_14; unsigned __int16 field_16; struct_12A42 *ptr_field_18; struct_1B1A9 *ptr_field_20; } **ptr_ptr_field_0; unsigned __int64 dat_field_8; unsigned __int64 dat_field_10; struct_12A0E *ptr_field_28; ... __int32 dat_field_74; ... } </pre>
--	--

(a) Ground truth

(b) By OSPREY

Figure 3.27. Reconstructed Symbols

structure). OSPREY can successfully recover the `server_socket` structure. In fact as shown in Figure 3.27a and 3.27b, OSPREY can precisely recover the multiple layers of structure nesting and all the pointer fields. Note that `server_socket_array` is an inner structure type without any outer reference. The recovery of the structure can substantially improve the readability of the decompiled code. See lines 19-20 in Figure 3.26a. Without the recovered information, we can only learn there are a memory access with complex addressing. With the recovered field and array accesses, we have much more semantic information.

3.7.2 Harden Stripped Binary

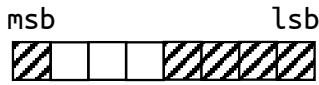
Exposing potential memory bugs is very important for vulnerability detection. Address sanitizer (ASAN) [77], a tripwire-based memory checker, can be used to increase the likelihood of triggering a crash when a memory corruption occurs. The principle of ASAN is to insert redzones at the border of variables. Program crashes whenever an out-of-bound access touches the redzone. The effectiveness of ASAN is determined by the accuracy of identifying



```

01. typedef struct node {
02.     long data[0x20];
03.     struct node *next;
04. } node_t;
05.
06. void gee() {
07.     node_t *p =
08.         malloc(sizeof(node_t));
09.     for (int i=0; i<=0x20, i++)
10.         p->data[i] = 0;

```

(a) Simplified example of CVE-2019-12802, which has an out-of-bound memory access for array data inside structure `node_t`.



-  Bits used by standard libasan.so.4.
-  Bits used by OSPREY enhancement.

(b) Address sanitizer maps 8 bytes of the application memory into 1 byte of the shadow memory named `shadow byte`. However, only 5 bits of each shadow byte are used in standard libasan.so.4.

```

[A] xor rcx, rcx
[B] cmp ecx, 0x20
[C] jg <ret_gee>
[C1] lea rdi, [rax+rcx*8]
[C2] shr rdi, 3
[C3] mov dil, [rdi+SHADOW_BASE]
[C3a] mov rsi, rdi
[C3b] and sil, 0x30
[C3c] cmp sil, 0x30
[C3d] jz <asan_report_error>
[C3e] and dil, 0x8F
[C4] test dil, dil
[C5] jnz <asan_report_error>
[D] mov [rax+rcx*8], 0
[E] inc ecx
[F] jmp B

```

(c) Assemble code for line 8 and line 9 in Figure 3.28a. Lines A, B, C, D, E, F are the original assemble code, lines C1, C2, C3, C4, C5 are instrumented by *RetroWrite*, and lines C3a, C3b, C3c, C3d, C3e are instrumented by our enhancement. Only our instrumentation can report CVE-2019-12802.

Figure 3.28. Field-level binary ASAN instrumentation for CVE-2019-12802.

the variable borders, which is very challenge if source code or debugging information is not available. The state-of-the-art binary-level ASAN solution (*RetroWrite* [78]) conducts very coarse-grained border identification. Specifically, for an allocated heap region, redzones are only inserted before and after the region, not between the variables/fields within the region. This may degrade the effectiveness of ASAN. Take CVE-2019-12802 [79] as an example. It is an out-of-bound vulnerability whose simplified code is shown in Figure 3.28a. The vulnerability occurs at line 9, in which there is an out-of-bound memory access for array `data` inside the `node_t` structure. *RetroWrite* does not insert redzone code within the `node_t` structure, hence cannot detect the vulnerability.

We strengthen *RetroWrite* to take in our reconstructed symbol information such that corruptions internal to a structure can be detected. Specifically, we aim to prevent scalar variables from being accessed by any array instruction. To avoid false warnings and offer a strong (probabilistic) guarantee, we carefully define *scalar variables* and *array instruction*. We define v as a scalar variable, if $P(\text{Scalar}(v)) > 0.99 \wedge \neg(\exists(a_1, a_2), \text{ s.t. } (a_1 \leq v.a \leq a_2) \wedge (P(\text{Array}(a_1, a_2)) > 0.01))$. Similarly, we define i as an array instruction, if $\forall v : \text{Accessed}(i, v), \exists(a_1, a_2), \text{stAccessMultiChunks}(i, v.a.r) \wedge (a_1 \leq v.a \leq a_2) \wedge (P(\text{Array}(a_1, a_2))) > 0.99$. We leverage *RetroWrite* to instrument the target binary. For any memory access by an array instruction, besides the basic ASAN checks provided by *RetroWrite*, we additionally check it is accessing a scalar variable.

Figures 3.28c and 3.28b present the details of our implementation. Lines [A] [B] [C] [D] [E] [F] in Figure 3.28c are the original assembly code for line 8-9 in Figure 3.28a, where `rcx` in line [B] stores the value of `i` and `rax+rcx*8` in line [D] stores the address of `p->data[i]`. Lines [C1] [C2] [C3] [C4] [C5] are instrumented by *RetroWrite*. They first get the target address of instruction [D] (line [C1]), read its shadow value (`dil`) from the corresponding shadow memory (lines [C2] [C3]), and validate the shadow value (lines [C4] [C5]). *RetroWrite*'s ASAN is based on the standard `libasan.so.4`. Hence it directly invokes `asan_report_error` to report errors. An interesting observation is that, even though `libasan.so.4` uses one byte to store shadow value, only 5 bits of the byte are used, as shown by the shadow value layout in Figure 3.28b. This allows us to store more meta information using the remaining 3 bits. In our case, we use one bit to record whether the memory stores a scalar variable. After that, we instrument more validation instructions for array instructions. Lines [C3a] [C3b] [C3c] [C3d] [C3e] are added by OSPREY, for array instruction [D]. The instrumentation validates whether the accessed memory stored a scalar variable. As such, the mentioned CVE can be successfully detected. The instrumented code does not cause any false warnings when executed on normal test cases. Note that although probabilistic guarantees may not be strong enough for production systems, they make perfect sense for vulnerability detection, in which rare false warnings are acceptable.

3.8 Summary

We develop a novel probabilistic variable and data structure recovery technique for stripped binaries. It features using random variables to denote the likelihood of recovery results such that a large number of various kinds of hints can be organically integrated with the inherent uncertainty considered. A customized and optimized probabilistic constraint solving technique is developed to resolve these constraints. Our experiments show that our technique substantially outperforms the state-of-the-art and improves two downstream analysis.

4. ITERATIVE REFINEMENT: EFFECTIVE AND EFFICIENT BINARY-ONLY FUZZING

Fuzzing stripped binaries poses many hard challenges as fuzzers require instrumenting binaries to collect runtime feedback for guiding input mutation. However, due to the lack of symbol information, correct instrumentation is difficult on stripped binaries. Existing techniques either rely on hardware and expensive dynamic binary translation engines such as QEMU, or make impractical assumptions such as binaries do not have inlined data. We observe that fuzzing is a highly repetitive procedure providing a large number of trial-and-error opportunities. As such, we propose a novel incremental and stochastic rewriting technique STOCHFuzz that piggy-backs on the fuzzing procedure. It generates many different versions of rewritten binaries whose validity can be approved/disapproved by numerous fuzzing runs. Probabilistic analysis is used to aggregate evidence collected through the sample runs and improve rewriting. The process eventually converges on a correctly rewritten binary. We evaluate STOCHFuzz on two sets of real-world programs and compare with five other baselines. The results show that STOCHFuzz outperforms state-of-the-art binary-only fuzzers (e.g., *e9patch*, *ddisasm*, and *RetroWrite*) in terms of soundness and cost-effectiveness and achieves performance comparable to source-based fuzzers. STOCHFuzz is publicly available [80].

4.1 Introduction

Grey-box fuzzing [10, 50, 81, 82] is a widely used security testing technique that generates inputs for a target program to expose vulnerabilities. Starting from some *seed inputs*, a fuzzer repetitively executes the program while mutating the inputs. The mutation is usually guided by coverage information. For instance, a popular strategy is that input mutations leading to coverage improvement are considered important and subject to further mutations. As such, existing fuzzing engines rely on instrumentation to track code coverage. Typically, they leverage compilers to conduct instrumentation before fuzzing when source code is available. However in many cases, only binary executables are available. Various techniques have been

developed to support fuzzing applications without source code. We call them *binary-only fuzzing* techniques.

Existing binary-only solutions fall into three categories: (1) leveraging hardware support, (2) leveraging on-the-fly dynamic binary rewriting, and (3) relying on offline static binary rewriting. The first category makes use of advanced hardware support such as Intel PT [83] to collect runtime traces that can be post-processed to acquire coverage information. Such traces record individual executed basic blocks, which are generated at a very high rate, and hence require substantial efforts to process. In addition, it is difficult to collect runtime information other than control-flow traces. The second kind uses dynamic rewriting engines such as QEMU [84] and PIN [85], which instrument a subject binary during its execution. They trap execution of each new basic block and rewrite it on the fly. The rewritten basic block is then executed. The method is sound but expensive due to the heavyweight machinery (4-5 times slower than source based fuzzing according to our experiment in Section 4.6). The third kind instruments the binary just once before the whole fuzzing process. However, sound static binary rewriting is an undecidable problem [86] due to the lack of symbol information. It entails addressing a number of hard challenges such as separating code and data, especially inlined data [70, 87], and identifying indirect jump and call targets [46, 64]. Existing solutions are either based on heuristics and hence unsound [64, 88], or based on restricted assumptions such as no inlined data is allowed [89] and relocation information must be available [78]. However, these assumptions are often not satisfied in practice. According to our experiment in Section 4.6, a number of state-of-the-art solutions, such as *e9patch* [89] and *ddisasm* [64] fail on real-world binaries.

We observe that fuzzing is a highly repetitive process in which a program is executed for many times. As such, it provides a large number of chances for trial-and-error, allowing rewriting to be incremental and progress with increasing accuracy over time. We hence propose a novel *incremental* and *stochastic* rewriter that piggy-backs on the fuzzing procedure. It uses probabilities to model the uncertainty in solving the aforementioned challenges such as separating data and code. In other words, it does not require the binary analysis to acquire sound results to begin with. Instead, it performs initial rewriting based on the uncertain results. The rewritten binary is very likely problematic. However, through a

number of fuzzing runs, the technique automatically identifies the problematic places and repairs them. The process is stochastic. It does not use a uniform rewritten binary. Instead, it may rewrite the binary differently for each fuzzing run by drawing samples from the computed probabilities. It randomly determines if bytes at some addresses ought to be rewritten based on the likelihood that the addresses denote an instruction. As such, the problematic rewritings are distributed and diluted among many versions, allowing easy fault localization / repair and ensuring fuzzing progress. Note that if a binary contains too many rewriting problems, the fuzzer may not even make reasonable progress, significantly slowing down the convergence to precise rewriting. In contrast, during stochastic rewriting, while some versions fail at a particular place, many other versions can get through the place (e.g., as they do not rewrite the place), which in turn provides strong hints to fix the problem. The probabilities are updated continuously across fuzzing runs as our technique sees more code coverage and fixes more rewriting problems, affecting the randomly rewritten versions. At the end, the uncertainty is excluded when enough samples have been seen, and the process converges on a stable and precisely rewritten binary.

Our contributions are summarized as follows.

- We propose a novel incremental and stochastic rewriting technique that is particularly suitable for binary-only fuzzing. It piggy-backs on fuzzing and leverages the numerous fuzzing runs to perform trial-and-error until achieving precise rewriting.
- The technique is facilitated by a lightweight approach that determines the likelihood of each address denoting a data byte. We formally define the challenge as a probabilistic inference problem. However, standard inference algorithms are too heavyweight and not sufficiently scalable in our context, which requires recomputing probabilities and drawing samples during fuzzing. We hence develop a lightweight approximate algorithm.
- We develop a number of additional primitives to support the process, which include techniques to automatically locate and repair rewriting problems.

<u>.CODE0:</u>				
0 : <code>mov rax, 13</code>	0 : <code>mov rax, 13</code>	<code>rax</code>	13	.CODE2-.DATA
7 : <code>mov [rax], rax</code>	7 : <code>mov [rax], rax</code>	[rax]	13	.CODE2-.DATA
10: <code>lea r8, [rip+8]</code>	10: <code>lea r8, [rip+8]</code>	r8	25	.DATA
17: <code>mov edx, [r8]</code>	17: <code>mov edx, [r8]</code>	rdx	4	.CODE1-.DATA
20: <code>add rdx, r8</code>	20: <code>add rdx, r8</code>	rdx	29	.CODE1
23: <code>jmp rdx</code>	23: <code>jmp rdx</code>	<code>jmp</code>	<u>.CODE1</u>	-
<u>.DATA:</u>				
25: <code>.int 4</code>				
<u>.CODE1:</u>				
29: <code>mov r9, [rax]</code>	29: <code>mov r9, [rax]</code>	r9	13	.CODE2-.DATA
32: <code>add r8, r9</code>	32: <code>add r8, r9</code>	r8	38	.CODE2
35: <code>jmp r8</code>	35: <code>jmp r8</code>	<code>jmp</code>	<u>.CODE2</u>	-
<u>.CODE2:</u>				
38: <code>mov rax, 60</code>	38: <code>mov rax, 60</code>	rax	60	-
45: <code>syscall</code>	45: <code>syscall</code>	-	-	-

Figure 4.1. Motivation example

- We develop a prototype STOCHFuzz [90] and evaluate it on the Google Fuzzer Test Suite [91], the benchmarks from *RetroWrite* [78], and a few commercial binaries. We compare it with state-of-the-art binary-only fuzzers *e9patch* [89], *ptfuzzer* [92], *ddisam* [64], *afl-qemu* [93] and *RetroWrite* [78] and also with source based fuzzers *afl-gcc* [50] and *afl-clang-fast* [94]. Our results show that STOCHFuzz outperforms these binary-only fuzzers in terms of soundness and efficiency, and has comparable performance to source based fuzzers. For example, it is 7 times faster than *afl-qemu*, and successfully handles all the test programs while other static binary rewriting fuzzers fail on 12.5–37.5% of the programs. Our fuzzer also identifies zero-days in commercial binaries without any symbol information. We have conducted a case study in which we port a very recent source based fuzzing technique IJON [95] that tracks state feedback in addition to coverage feedback, to support binary-only fuzzing. It demonstrates the applicability of STOCHFuzz. Our system and benchmark corpora are publicly available [80].

4.2 Motivation

In this section, we use an example to illustrate the limitations of existing binary-only fuzzing techniques and motivate ours. Figure 4.1 presents a piece of assembly code for illustration purpose (its functionality is irrelevant). The right side of the figure depicts its execution trace - where the executed instructions, destination registers, and evaluation results are listed in the first three columns, respectively. The last column presents the related section(s) if the evaluated result is address relevant. For example, the value 25 generated by the instruction at address 10 denotes an address in the `.DATA` section while the value 29 generated by the instruction at address 20 denotes an address in `.CODE1`.

As shown, the snippet consists of three code sections (i.e., `.CODE0`, `.CODE1`, and `.CODE2`) and an interleaved data section `.DATA`. The first two instructions (at addresses 0 and 7) in `.CODE0` load a constant 13 to `rbx`, and then store it in a memory location denoted by `[rax]`. The constant 13 denotes the offset between the `.CODE2` section and the `.DATA` section, i.e., $38-25=13$, and will be used later in addressing. The three instructions at addresses 10, 17, and 20 calculate the address of label `.CODE1`. Specifically, `r8` is first set to the address of `.DATA` via a PC-related `lea` instruction. At address 17, an integer 4 representing the offset between labels `.CODE1` and `.DATA` is loaded from the memory address denoted by `[r8]` (i.e., address 25) to `edx`, which consequently updates `rdx`. Next, `r8` is added to `rdx`. The resulting `rdx` denotes the address of `.CODE1`. The subsequent instruction at 23 triggers an indirect jump to label `.CODE1`. The next two instructions at addresses 29 and 32 determine the target of the indirect jump at address 35 (i.e., `.CODE2`) by loading the offset 13 from `[rax]` and adding it to the address of `.DATA` stored in `r8`. A `syscall` is invoked subsequently once the indirect jump is triggered. Observe that the code snippet has inlined data, indirect jumps, and complex address computation, which pose substantial challenges to existing binary-only fuzzers.

4.2.1 Limitations of Existing Technique

Recall that fuzzers need to collect runtime feedback such as code coverage to guide input mutation. For binary-only fuzzers, such feedback can be captured by a technique in one of

Table 4.1. Summary of different *binary-only fuzzing* instrumentation techniques, along with compiler instrumentation (*afl-gcc* and *afl-clang-fast*)

Tool	Prerequisite				Support		Soundness	Efficiency
	<u>A1</u>	<u>A2</u>	<u>A3</u>	<u>A4</u>	<u>S1</u>	<u>S2</u>		
afl-gcc	Require Source Code						Sound	A
afl-clang-fast	Require Source Code						Sound	A+
ptfuzzer [92]	-	-	-	-	Y	N	Sound	C
afl-qemu	-	-	-	-	Y	Y	Sound	D
afl-dyninst [88]	-	-	✓	-	Y	Y	Unsound	A
e9patch [89]	-	-	✓	✓	Y	Y	Sound	B
RetroWrite [78]	✓	✓	✓	✓	N	Y	Unsound	A
ddisasm [64]	-	-	-	-	Y	Y	Unsound	A
StochFuzz	-	-	-	✓	Y	Y	Sound	A
	-	-	-	-	Y	Y	Prob sound	A

the following three categories: (1) hardware-assisted tracing, (2) dynamic binary instrumentation, and (3) static binary rewriting. In Table 4.1, we summarize the characteristics of existing techniques. Column 1 lists these techniques, with the first two being source-based AFL fuzzers using *gcc* and *clang* compilers, *ptfuzzer* using hardware-assisted tracing, *afl-qemu* using dynamic instrumentation, and the others including ours using static binary rewriting. Columns 2-5 are the assumptions made by these tools, where ✓ denotes that a specific precondition is required. A1 denotes that the binary has symbol and relocation information, A2 denotes that the binary is Position Independent, A3 denotes that all instruction boundaries are correctly identified by upstream disassembler, and A4 denotes that the binary does not contain any inlined data. Columns 6 and 7 show whether C++ programs and other runtime feedback beyond coverage are supported, respectively. S1 denotes that the tool supports binaries compiled from C++ programs, and S2 denotes that the tool supports collecting other runtime information than coverage. Column 8 denotes the soundness guarantee which means if the technique guarantees to rewrite the binary properly and collect the right feedback, and column 9 denotes fuzzing efficiency with A+ the best. Note that the soundness of STOCHFuzz can be guaranteed when there is no inlined data, and probabilistically guaranteed otherwise.

Hardware-assisted Tracing. Modern processors offer a mechanism that captures software execution information using dedicated hardware [83]. PTFuzzer [92] leverages this feature to collect code coverage for binary-only fuzzing. For instance, after executing the code in Figure 4.1, two control transfers are recorded, i.e., from 23 to 29 and from 35 to 38. Based on the information, PTfuzzer subsequently recovers the execution path and hence the coverage. Other hardware-assisted fuzzers operate similarly [96, 97]. The performance of these approaches is limited by the costly trace post-processing ($4\times$ slower than *afl-clang-fast* according to our experiments). Additionally, hardware-assisted fuzzing cannot capture other runtime feedback than coverage [82, 95].

Dynamic Instrumentation. Dynamic instrumentation translates and instruments the binary during execution [84, 85]. Although it is an attractive solution due to its sound instrumentation, the on-the-fly translation/instrumentation incurs relatively higher runtime overhead compared to other approaches. *Afl-qemu*, to the best of our knowledge, is among the best-performing binary-only fuzzers based on dynamic instrumentation. It still incurs significant overhead ($5\times$ slower than *afl-clang-fast* according to our experiments). Other approaches in this category, including *afl-pin* [98] and *afl-dynamorio* [99], induce even higher overhead.

Static Binary Rewriting. Static rewriting utilizes binary analysis to disassemble and rewrite the binary before execution. Unfortunately, it is still a hard challenge to rewrite stripped binary with soundness guarantee. Existing solutions often make unsound assumptions about the target binary which may lead to runtime failures.

Afl-dyninst [88], a trampoline-based approach built upon traditional disassembly techniques, assumes the upstream disassemblers can correctly identify all the instructions. However, such assumption may not hold in practice due to code and data interleavings [78, 89]. Figure 4.2 demonstrates how the code example in Figure 4.1 breaks its assumption, where the red box shows corrupted code, and the yellow box shows missing code. The left of Figure 4.2 shows that a *linear disassembly*, which decodes all bytes consecutively, is confused by address 25, the inlined data byte. *Recursive disassembly*, on the other hand, avoids this

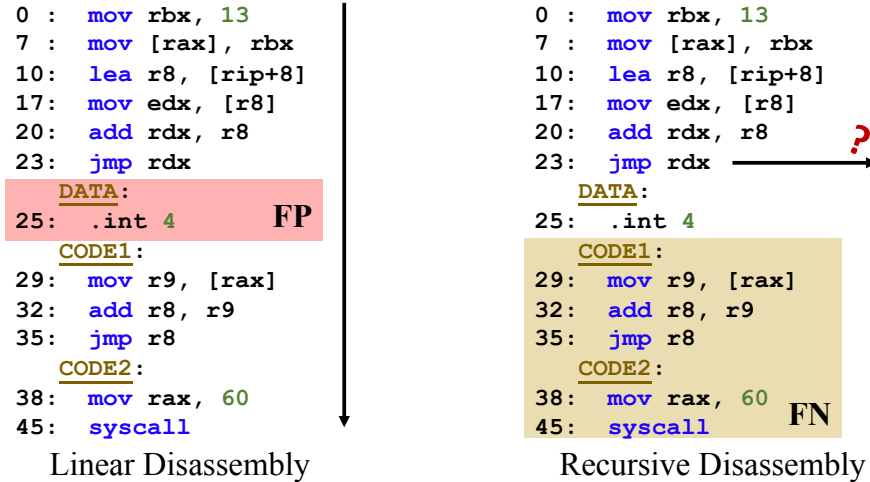


Figure 4.2. Limitations of disassembly methods

problem by disassembling instructions following control flow. But it fails to resolve the target of the indirect jump at address 23, missing the code from address 29 to 45.

E9patch [89] makes the same assumption as *afl-dyninst*, and additionally assumes there is no inlined data. With these assumptions, *e9patch* specially engineers jumps that can safely overlap with other instructions. As such, it can insert trampolines without sacrificing the correctness of rewriting. In addition, it uses a sophisticated virtual address space layout for the instrumented binary, which on the other hand might make it susceptible to a large number of cache misses and additional overhead in process forking[100].

RetroWrite [78] is a reassembly technique for *Position Independent Code* (PIC). It converts address related immediate values in the binary to symbols (called *symbolization*) such that they can be easily relocated after instrumentation. For example in Figure 4.3, the “`lea r8, [rip+8]`” instruction at address 10 is translated as “`lea r8, [L25]`”, because *RetroWrite* recognizes that `rip+8` denotes a reference in the code space and needs to be symbolized. As such, it could be properly relocated after instrumentation. However, sound static symbolization is provably undecidable [86] in general. *RetroWrite* consequently makes strong assumptions such as the requirement of relocation information and the exclusion of C++ exception handlers. However, even if these requirements were satisfied, the soundness of *RetroWrite* still could not be guaranteed due to the need of sound memory access

<pre> 0 : mov rbx, 13 7 : mov [rax], rbx 10: lea r8, [rip+8] 17: mov edx, [r8] 20: add rdx, r8 23: jmp rdx DATA: 25: .int 4 CODE1: 29: mov r9, [rax] 32: add r8, r9 35: jmp r8 CODE2: 38: mov rax, 60 45: syscall </pre>	<p>Reassemble & Instrument</p>	<pre> 0 : [afl trampoline] 10: mov rbx, 13 17: mov [rax], rbx 20: lea r8, [L25] # r8=35(.L25) 27: mov edx, [r8] # rdx=.L29-.L25 30: add rdx, r8 # rdx=.L29 33: jmp rdx # correct(.L29) L25: 35: .int .L29-.L25 L29: 39: [afl trampoline] ← 49: mov r9, [rax] # r9=13 52: add r8, r9 # r8=45 55: jmp r8 58: mov rax, 60 65: syscall </pre>
---	--	--

Figure 4.3. Reassembly in *RetroWrite*. It crashes as the constant 13 in red circle is not properly symbolized.

reasoning. In the right side of Figure 4.3, recognizing that the constant 13 in the first instruction “mov rbx, 13” is an address offset (and needs symbolization) is challenging, due to the long sequence of complex memory operations between this instruction and the final address de-reference at 55, which ultimately discloses constant 13 is an address offset. In the example, *RetroWrite* misclassifies 13 as a regular value. As a result, it is not symbolized. Ideally, it should be symbolized to .L38-.L25, which would be concretized to $58-35=23$ after instrumentation. As a result, *RetroWrite* crashes on the binary. A recent study [64] shares the same concern.

Ddisasm [64] is a state-of-the-art reassembly technique. Rather than making assumptions about target programs, it relies on a large set of reassembly heuristics such as instruction patterns. These heuristics, although comprehensive, have inherent uncertainty and may fail in many cases.

4.2.2 Our Technique

Our technique is inspired by two important insights. *First insight: while grey-box fuzzers continuously mutate inputs across test runs, they may as well be enhanced to mutate the program on-the-fly. As such, disassembly and static rewriting (which are difficult due to the*

lack of symbol information and difficulties in resolving indirect jumps/calls offline) can be incrementally performed over time.

Example. We use case A in the first row of Figure 4.4 to demonstrate how our technique leverages the first insight. The workflow consists of four steps, an initial patching step prior to fuzzing (step ①) and three incremental rewriting steps during fuzzing (steps ②, ③, and ④).

In the snippet to the left of ①, the code sections are filled with a special one-byte `hlt` instruction, which will cause a segfault upon execution. A segfault by a `hlt` instruction indicates that the system has just discovered a code region that has not been properly disassembled or rewritten such that incremental rewriting should be performed. We will explain later how we separate code and data in the first place (as only code is replaced with `hlt` in the snippet). The separation of the two does not have to be precise initially and our stochastic rewriting (discussed later) can gradually improve precision over the numerous fuzzing runs. For instance, the execution of initial patched code is terminated by the `hlt` at address 0, indicating a new code region. For easy description, we call such segfaults *intentional crashes*.

The next step (incrementally) rewrites all the addresses that can be reached along direct control flow from the address where the intentional crash happens. Specifically, `STOCHFuzz` places the rewritten code in a new address space, called the *shadow space*; it further redirects all the direct jumps and calls to their new targets in the shadow space by patching immediate offsets; and since data sections are retained in their original space, any PC-dependent data references need to be properly patched too. At last, `STOCHFuzz` inserts a jump instruction at the crash address to direct the control flow to the shadow space. In the code snippet in between ① and ②, given the crash address 0, `STOCHFuzz` disassembles the instructions from addresses 0 to 23 (highlighted in green shade). These instructions are then rewritten in the shadow space starting from address 90. Specifically, an `afl trampoline` is inserted at the beginning to collect coverage information, and the original “`lea r8, [rip+8]`” instruction (at address 10) is rewritten to “`lea r8, [rip-92]`” (at address 110) to ensure the data reference occurs at the original address. `STOCHFuzz` inserts a “`jmp 90`” instruction at 0 to

transfer the control flow. Then, the fuzzer continues fuzzing with the new binary and the incremental rewriting is invoked again if other intentional crashes occur (e.g., steps ② and ③). \square

A prominent challenge is to separate code and data in executables, especially when inlined data are present. Due to the lack of symbol information, it is in general an undecidable problem [86]. Heuristics or learning based solutions [64, 101] are inevitably unsound. Data may be recognized as instructions and replaced with `hlt`. As a result, the program may execute with corrupted data which may or may not manifest themselves as crashes. Corrupted states may lead to bogus coverage and problematic test results. On the other hand, instructions may be recognized as data and hence not replaced with `hlt`. Consequently, these instructions are invisible to our system and not instrumented.

The following second insight allows us to address the aforementioned problem. *Second insight: fuzzing is a highly repetitive process that provides a large number of opportunities for trial-and-error. That is, we can try different data and code separations, which lead to different instrumented executables, in different fuzzing runs. Over time, an increasing number of samples can be collected, allowing us to achieve the precise separation and correct rewriting.* There are two challenges that we need to overcome in order to leverage the insight. First, we need to distinguish exceptions caused by rewriting errors (introduced by our trial-and-error) and by latent bugs in the subject program. We call both *unintentional crashes* (to distinguish from *intentional crashes* by `hlt`). We also need to pinpoint and repair rewriting errors, i.e., data bytes misclassified as code (and undesirably replaced with `hlt`), and vice versa. We call it the *self-correction requirement*. Second, an executable cannot contain too many rewriting errors. Otherwise, the fuzzing runs of the executable can hardly make progress (as it continues to crash on these errors one after another). Note that we rely on the fuzzer’s progress to collect more and more samples to correct our rewritings. We call it the *progress requirement*.

We therefore propose a novel *stochastic rewriting technique* that piggy-backs on the fuzzing procedure. At first, the technique performs probabilistic inference to compute the likelihood of individual bytes in the original address space belonging to data (or code). Such probabilities are computed based on various hints, such as register definition-use relations

that often indicate instructions and consecutive printable bytes that often suggest data. Details of the probabilistic inference can be found in Section 4.3.1. Since these hints are inherently uncertain (e.g., printable bytes may not be data), we use probabilities to model such uncertainty. Based on the computed probabilities, STOCHFuzz randomly generates a rewritten version for each fuzzing run. In a random version, the bytes replaced with `hlt` are determined by sampling based on their computed probabilities. For instance, a byte with a high probability of being code is more likely replaced with `hlt`. When a segfault is observed, STOCHFuzz determines if it is caused by a rewriting error, by running the failure inducing input on a binary with all the uncertain rewritings removed and observing if the crash disappears. If so, delta debugging [102], a binary-search like debugging technique, is used to determine the root cause rewriting. Over time, the corrected rewritings, together with the new coverage achieved during fuzzing, provide accumulating hints to improve probabilistic inference and hence rewriting. Note that the proposed solution satisfies the two aforementioned requirements: the rewriting errors are distributed in many random versions such that the fuzzer can make progress in at least some of them; and they can be automatically located and repaired.

Example Continued. We use case B (the lower box) in Figure 4.4 to illustrate stochastic rewriting. At the beginning (the snippet to the left of ① in case B), STOCHFuzz computes the initial probabilities (of being data bytes) as shown to the left of the individual addresses. For example, a definition-use relation between addresses 0 and 7 caused by `rbx` decreases their probability of being data. Assume in a random binary version the addresses with color shades are replaced by `hlt`, with the yellow ones being the correct replacements as they denote instructions and the red one erroneous since a data byte is replaced with a `hlt`. The binary is executed and then an intentional crash is encountered at address 0. In the snippet to the right of ①, besides the incremental rewriting mentioned in case A, STOCHFuzz also performs probability recalculation which updates the probabilities based on the new hints from the execution. Intuitively, as address 0 is code, all addresses (in green shade) reachable from the instruction along control flow must be code. We say that they are “*certainly code*” and their probabilities are set to 0. The probabilities of remaining addresses are updated

and new random binaries are generated. *In practice, many of the misclassified bytes such as 25 are proactively fixed by these new hints and updated probabilities, without causing any crashes or even being executed.* This illustrates the importance of the aforementioned progress requirement.

However to make our discussion interesting, we assume 25 (i.e., the data byte) and 38 are still replaced in the new version (i.e., the snippet to the right of ①). During execution, since the data at 25 is corrupted, a wrong target address value is computed for `rdx` in the jump instruction at 123, causing a segfault. The diagnosis and self-correction procedure is hence invoked (steps ②-⑤). Specifically, the binary cleaning step ② removes all the rewritings at uncertain addresses (in yellow or red shades) and re-executes the program (to the right of ②). The crash at address 123 disappears, indicating the crash must be induced by a rewriting error. STOCHFuzz uses delta debugging and generates two binaries, one with only 25 replaced (i.e., the snippet to the left of ④) and the other with 38 replaced (i.e., the snippet to the left of ⑤). The former crashes at the same address 123 whereas the latter crashes at 38 (and hence an intentional crash). As such, STOCHFuzz determines that the rewriting of address 25 is wrong and fixes it by marking it as “*certainly data*” (i.e., with probability 1.0) in the version to the right of ⑤. This new hint leads to probability updates of other addresses (e.g., 29 and 32). The procedure continues and eventually all addresses have certain classification (i.e., all in green shade) and the program is properly rewritten. □

4.3 System Design

The architecture of STOCHFuzz is shown in Figure 4.5. It consists of five components: the probability analyzer, the incremental and stochastic rewriter, the program dispatcher, the execution engine, and the crash analyzer. The probability analyzer computes a probability for each address in the given binary to indicate the likelihood of the address denoting a data byte. The rewriter rewrites the binary in different forms by sampling based on the computed probabilities. The program dispatcher selects a rewritten version to execute, either randomly for a normal execution request or strategically for root cause diagnosis. The execution engine, a variant of AFL [50], executes a given binary and monitors for crashes.

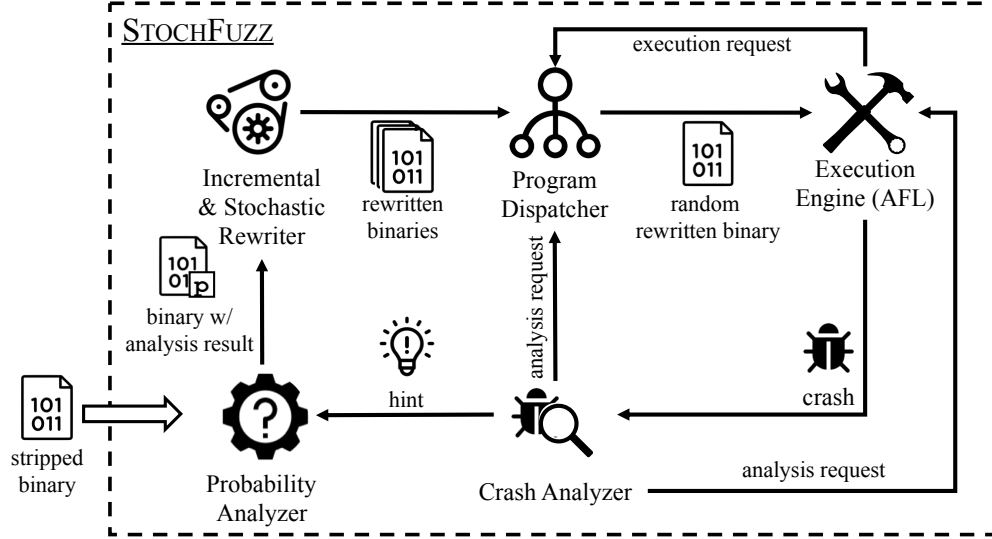


Figure 4.5. Architecture

The crash analyzer triggers incremental rewriting when it determines a crash is intentional; otherwise, it analyzes the root cause and automatically repairs it if the cause is a rewriting error.

STOCHFuzz has three typical workflows. *Case one* is the most common. It is similar to the standard AFL. Specifically, the execution engine sends a request to the program dispatcher for a binary. The dispatcher randomly selects a rewritten binary (from its pool), which is then executed by the engine. The binary subsequently exits normally without any crash.

In *case two*, the execution is terminated by an intentional crash (i.e., a `hlt` instruction). The crash is reported to the crash analyzer, which identifies the new code coverage indicated by the crash and analyzes the newly discovered code to collect additional hints for distinguishing data and code. The hints are passed on to the probability analyzer, which recomputes the probabilities and invokes the incremental rewriter to generate new binaries.

In *case three*, the execution is terminated by an unintentional crash (i.e., a crash not caused by `hlt`). To verify whether the crash is triggered by some rewriting error, the crash analyzer notifies the program dispatcher to send a binary that has all uncertain rewritings removed for execution. If the previous crash persists, it must be caused by a latent bug in

the original program. Otherwise, the crash is caused by rewriting error. The crash analyzer further performs delta-debugging to locate the root cause and repairs it. The repair is passed on as a hint to the probability analyzer and triggers probabilities updates and generation of new binaries. In the remainder of this section, we discuss details of the components.

4.3.1 Probability Analyzer

This component computes the probabilities of each address denoting data or code. Initially (before fuzzing starts), it computes the probabilities based on the results of a simple disassembler that we only use to disassemble at *each address in the binary*. During fuzzing, with new observations (e.g., indirect call and jump targets) and exposed rewriting errors, it continuously updates probabilities until convergence. It models the challenge as a *probabilistic inference* problem [65]. Specifically, random variables are introduced to denote individual addresses’ likelihood of being data or code. Prior probabilities, which are usually predefined constants as in the literature [59, 66–68], are associated with a subset of random variables involved in observable features (e.g., definition-use relations that suggest likely code). Random variables are correlated due to program semantics. The correlations are modeled as probabilistic inference rules. Prior probabilities are propagated and aggregated through these rules until convergence using probabilistic inference algorithms, yielding *posterior probabilities*. In the following, we explain how we define the problem and introduce our lightweight solution.

Definitions and Analysis Facts. As shown in the top of Figure 4.7, we use a to denote an address, c a constant, and r a register. The bottom part of Figure 4.7 presents the analysis facts directly collected from the binary. These facts are deterministic (not probabilistic). $Inst(a, c)$ denotes that the c bytes starting from address a can be encoded as a valid instruction. $ExplicitSucc(a_1, a_2)$ denotes the instruction at address a_2 is an explicit successor of the instruction at address a_1 along control flow. $RegWrite(a, r)$ denotes the instruction at a writes to register r . $RegRead$ denotes the read operation. $Str(a, c)$ denotes the c bytes starting from address a constitute a printable null-terminated string.

Initially, STOCHFuzz disassembles at each address and collects the analysis facts. It collects more facts than those in Figure 4.7. They are elided due to space limitations.

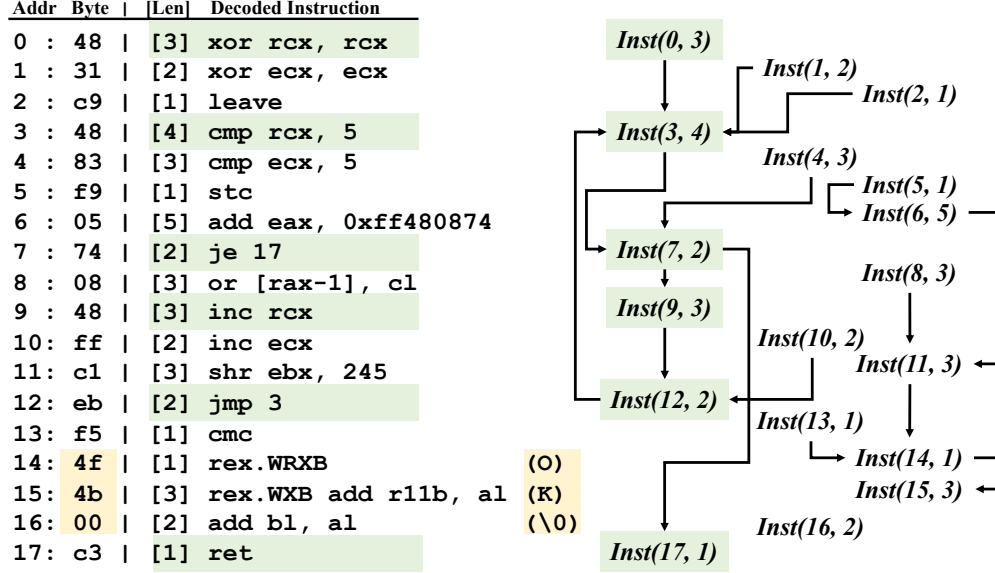


Figure 4.6. Universal Control-flow Graph (UCFG) Example. On the left, each address is disassembled (with the real instructions in green shade and the real data in yellow). The corresponding UCFG is in the right.

Example. In the left of Figure 4.6, STOCHFuzz disassembles starting from each (consecutive) address of a binary, with the first column showing the addresses, the second column the byte value at the address, the third column the instruction size, and the last column the instruction. For example, the first three bytes “48 31 c9” are disassembled to an `xor` instruction and the four bytes starting from address 3 are disassembled to a `cmp` instruction. We highlight the true instructions in green shade, and the true data, an “OK” string, in yellow shade, for discussion convenience. Note that STOCHFuzz does not assume such separation a priori. A simple sound binary analysis yields the following facts: $Inst(7, 2)$ because the instruction at 7 is “je 17” whose instruction size is 2, $ExplicitSucc(7, 17)$ as the instruction at 7 jumps to 17, $RegWrite(9, rcx)$, $RegRead(9, rcx)$, and $Str(14, 3)$. \square

Predicates. Next, we introduce a set of *predicates* that describe inference results. Different from facts that are deterministic, predicates may be uncertain. A random variable is hence associated with each uncertain predicate, denoting the likelihood of it being true. A subset of the predicates we use are presented in the top of Figure 4.8 with those having overline

$a \in \langle \text{Address} \rangle ::= \text{Integer} \quad c \in \langle \text{Constant} \rangle ::= \text{Integer}$ $r \in \langle \text{Register} \rangle ::= \{\text{rax}, \text{rbx}, \text{rcx}, \text{rdx}, \dots\}$

Inst(a, c) : the c bytes starting from address a can be disassembled as an inst
ExplicitSucc(a_1, a_2) / : the inst at a_2 is an *explicit* successor of the one at a_1
RegWrite(a, r) / ***RegRead***(a, r) : the inst at a writes/reads data into/from reg r
Str(a, c) : the c bytes starting from addr a can be interpreted as a printable string

Figure 4.7. Definitions for variables and analysis facts

uncertain. $\text{ExplicitReach}(a_1, a_2)$ denotes that address a_1 can reach a_2 along control flow. In Figure 4.6, the path $0 - 3 - 7$ leads to $\text{ExplicitReach}(0, 7)$. $\text{RegLive}(a_1, a_2, r)$ denotes that register r written by address a_1 is live before the instruction at a_2 . As such, we have $\text{RegLive}(9, 12, \text{rcx})$ in Figure 4.6. $\overline{\text{IsInst}(a)}$ denotes the likelihood of address a being code. $\overline{\text{IsData}(a)}$ is similar.

(Probabilistic) Inference Rules. In the bottom of Figure 4.8, we present a subset of our inference rules. Some of them are probabilistic (i.e., those involving uncertain predicates and having probability on the implication operator). Here, 1.0, 0.0, p_{inst} , p_{data} , and p_{prob} denote prior probabilities that are predefined constants. Rules ① and ② derive control flow relations. Intuitively, an instruction can always reach its explicit successor (rule ①), and if a_1 can reach a_2 , it can reach the successors of a_2 (rule ②). Rules ③, ④, and ⑤ are to derive definition-use relations. Specifically, rule ③ denotes that if an instruction writes/defines a register, the register is live before the successor. Rule ④ denotes propagation of register liveness, that is, if a register is live before an instruction and the instruction does not overwrite the register, it remains live after the instruction. Rule ⑤ states that if there is a definition-use relation between a_1 and a_2 , both addresses are likely code, with a prior probability p_{inst} . Rule ⑥ states that if an address is likely code, all the addresses reachable from the instruction (at the address) have at least the same likelihood of being code. Rule ⑦ states that all bytes in a printable null-terminated string are likely data. Rule ⑧ leverages the continuity property of data and states that if two data addresses are close

$\mathbf{ExplicitReach}(a_1, a_2) : a_1 \text{ can explicitly reach } a_2 \text{ along control flow}$ $\mathbf{RegLive}(a_1, a_2, r) : \text{register } r \text{ written by address } a_1 \text{ is live before address } a_2$ $\overline{\mathbf{IsInst}(a) / \mathbf{IsData}(a)} : \text{the content at address } a \text{ is an inst/data byte}$	
<hr/>	
①	$\mathbf{ExplicitSucc}(a_1, a_2) \rightarrow \mathbf{ExplicitReach}(a_1, a_2)$
②	$\mathbf{ExplicitReach}(a_1, a_2) \wedge \mathbf{ExplicitSucc}(a_2, a_3) \rightarrow \mathbf{ExplicitReach}(a_1, a_3)$
③	$\mathbf{RegWrite}(a_1, r) \wedge \mathbf{ExplicitSucc}(a_1, a_2) \rightarrow \mathbf{RegLive}(a_1, a_2, r)$
④	$\mathbf{RegLive}(a_1, a_2, r) \wedge \neg \mathbf{RegWrite}(a_2, r) \wedge \mathbf{ExplicitSucc}(a_2, a_3) \rightarrow$ $\mathbf{RegLive}(a_1, a_3, r)$
⑤	$\mathbf{RegLive}(a_1, a_2, r) \wedge \mathbf{RegRead}(a_2, r) \xrightarrow{p_{\text{inst}} \uparrow} \overline{\mathbf{IsInst}(a_1)} \wedge \overline{\mathbf{IsInst}(a_2)}$
⑥	$\overline{\mathbf{IsInst}(a_1)} \wedge \mathbf{ExplicitReach}(a_1, a_2) \xrightarrow{1.0} \overline{\mathbf{IsInst}(a_2)}$
⑦	$\mathbf{Str}(a_1, c) \wedge (a_1 \leq a_2 < a_1 + c) \xrightarrow{p_{\text{data}} \uparrow} \overline{\mathbf{IsData}(a_2)}$
⑧	$\overline{\mathbf{IsData}(a_1)} \wedge \overline{\mathbf{IsData}(a_2)} \wedge (a_1 \leq a_3 \leq a_2 < a_1 + D) \xrightarrow{p_{\text{prop}} \uparrow}$ $\overline{\mathbf{IsData}(a_3)}$
⑨	$\overline{\mathbf{IsInst}(a)} \xleftrightarrow{0.0} \overline{\mathbf{IsData}(a)}$

Figure 4.8. Predicates and (probabilistic) inference rules. The predicates with overline are uncertain and rules with probability on top of \rightarrow denote probabilistic inference.

enough, the addresses in between are likely data too. Rule ⑨ states that an address cannot be code and data at the same time.

Incremental Fact and Rule Updates. New information can be derived during fuzzing and allows facts and rules to be updated. Specifically, new code coverage would allow deriving new facts such $\mathbf{ExplicitSucc}(\dots)$ (e.g., newly discovered indirect control flow). When a rewriting error that replaces a data byte a with `hlt` is located, the corresponding predicate $\overline{\mathbf{IsData}(a)}$ is set to a 1.0 prior probability, meaning “*certainly data*”. $\overline{\mathbf{IsInst}(a)}$ can be similarly updated. These updates will be leveraged by probabilistic inference to update other random variables and eventually affect stochastic rewriting.

Probabilistic Inference by One-step Sum-product. The essence of probabilistic inference is to derive posterior probabilities for random variables by propagating and aggregating prior probabilities (or *observations*) following inference rules. A popular inference method is *belief propagation* [103] which transforms the random variables (i.e., the uncertain predi-

cates) and probabilistic inference rules to a *factor graph* [65, 104], which is bipartite graph containing two kinds of nodes, a *variable node* for each random variable and a *factor node* for each probabilistic inference rule. A factor can be considered a function over variables such that edges are introduced between a factor node to the variables involved in the rule. Prior probabilities are then propagated and aggregated through the factor graph by an algorithm like *sum-product* [104], which is an iterative message-passing based algorithm. In each iteration, each variable node receives messages about its distribution from the factors connected to the variable, aggregates them through a *product* operation and forwards the resulted distribution through outgoing messages to the connected factor nodes. Each factor receives messages from its variables and performs a marginalization operation, or the *sum* operation. The posterior probabilities of random variables can be derived by normalizing the converged variable values.

However, belief propagation is known to be very expensive, especially when loops are present [105]. Most existing applications handle graphs with at most hundreds of random variables and factors [59, 66–68]. However in our context, we have tens of thousands of random variables and factors (proportional to the number of bytes in the binary). Resolving the probabilities may take hours. We observe that the factor graph is constructed from program that has a highly regular structure. The rounds of sum and product operations in the factor graph can be simplified to non-loopy explicit operations along the program structure. We hence propose a *one-step sum-product* algorithm that has linear complexity. The algorithm constructs a *universal control flow graph* (UCFG) that captures the control flow relations between the instructions disassembled at all addresses. Note that the binary’s real control flow graph is just a sub-graph of the UCFG. *Observations* (i.e., deterministic facts and predicates that suggest data or code) are explicitly propagated and aggregated along the UCFG, instead of the factor graph. In the last step, a simplest factor graph is constructed for each address to conduct a one-step normalization (from the observations propagated to this address) to derive the posterior probability (of the address holding a data byte). The factor graphs of different addresses are independent, precluding unnecessary interference.

Universal Control Flow Graph. In UCFG, a node is introduced for each address in the binary regardless of code or data, denoting the one instruction disassembled from that address. Edges are introduced between nodes if there is explicit control flow between them. UCFG is formally defined as $G = (V, E)$, where $V = \{a \mid \exists c \text{ s.t. } Inst(a, c)\}$ and $E = \{(a_1, a_2) \mid ExplicitSucc(a_1, a_2)\}$. The right side of Figure 4.6 presents the UCFG for the binary on the left. Note that only the shaded sub-graph is the traditional CFG. After UCFG construction, STOCHFuzz identifies the *strongly connected components* (SCCs) in the UCFG (i.e., nodes involved in loops). A node not in any loop is an SCC itself. For example in Figure 4.6, $Inst(0, 3)$ itself is a SCC. $Inst(3, 4)$, $Inst(7, 2)$, $Inst(9, 3)$, and $Inst(12, 2)$ form another SCC. \square

One-step Sum-product. The overall inference procedure is described as follows. STOCHFuzz first performs deterministic inference (following deterministic rules such as rules (1)-(4)). The resulted deterministic predicates such as the antecedents in rules (5) and (7) are called *observations*, with the former a code observation (due to the definition-use relation) and the latter a data observation. Prior probabilities p_{inst} and p_{data} are associated with them, respectively.

STOCHFuzz starts to propagate and aggregate these observations using UCFG. Specifically, it uses a product operation to aggregate all the observations in an SCC (i.e., multiplying their prior probabilities), inspired by the sum-product algorithm that uses a product operation to aggregate information across factors. All the addresses within the SCC are assigned the same aggregated value. Intuitively, we consider all the addresses in an SCC have the same likelihood of being code because any observation within an SCC can be propagated to any other nodes in the SCC (through loop). The lower the aggregated value, the more likely the address being code. We say the belief is stronger. The aggregated observations are further propagated across SCCs along control flow, until all addresses have been reached.

Data observations are separately propagated, mainly following rule (8). Specifically, STOCHFuzz scans through the entire address space in order, if any two data observations are close to each other (less than distance D), the addresses in between are associated with a value computed from the prior probabilities of the two bounding observations.

After propagation, each address a has two values denoting the aggregated code observation and the aggregated data observation, respectively. A simple factor graph is constructed for a as shown in Figure 4.9. The circled node a is the variable node, representing the likelihood of a being data. It has two factor nodes \mathcal{F}_{code} and \mathcal{F}_{data} , denoting the aforementioned two values. According to the sum-product algorithm [104], the posterior probability of a is the normalized product of the two factors as shown in the bottom of the figure.

Algorithm 7 details the one-step sum-product inference procedure. O_{code} and O_{data} denote the aggregated code and data observation values for each address, respectively. Note that *a small value means strong belief*. Line 3 performs the deterministic inference. Line 9 identifies SCCs and transforms UCFG to a DAG of SCCs. Step 1 in lines 13-20 propagates code observations. Step 2 in lines 22-33 propagates data observations. The formula in line 28 is derived from a simple factor graph involving three variables (i.e., addresses i , i_{prev} , and j), and three factors (for $O_{data}[i_{prev}]$, $O_{data}[i]$, and rule (7)). Details are elided. Step 3 in lines 35-39 performs the one-step sum-product for each address. Lines 36 and 37 assign observation value 0.5 if there is no belief propagated to the address. The formula in line 38 is derived from that in Figure 4.9.

Comparison with Probabilistic Disassembly. In probabilistic disassembly [60], researchers use probabilistic analysis to disassemble stripped binaries. It computes probabilities for each address to denote the likelihood of the address belonging to an instruction. However, its problem definition and probability computation are ad-hoc. Its algorithm is iterative and takes tens of minutes to compute probabilities for a medium-sized binary. It has a lot of false positives (around 8%), i.e., recognizing data bytes as instructions. These make it unsuitable for our purpose. In contrast, we formulate the problem as probabilistic inference and propose an algorithm with linear complexity. Piggy-backing on fuzzing, STOCHFuzz can achieve precise disassembly and rewriting with probabilistic guarantees.

4.3.2 Incremental and Stochastic Rewriting

The rewriter is triggered initially and then repetitively when new code is discovered or rewriting errors are fixed. It rewrites instructions in the shadow space (for better instrumen-

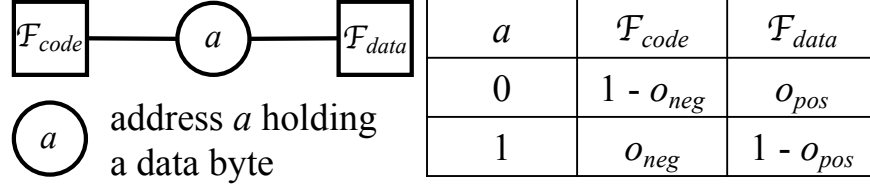
Algorithm 7 One-step Sum-product

INPUT:	B	binary indexed by address
OUTPUT:	$P[a] \in [0, 1]$	probability of address a holding a data byte
LOCAL:	$G = (V, E)$	$V = \{a \mid \exists c \text{ s.t. } Inst(a, c)\}$ $E = \{(a_1, a_2) \mid ExplicitSucc(a_1, a_2)\}$
	$O_{code}[a] \in [0, 1]$	aggregated code observations on address a
	$O_{data}[a] \in [0, 1]$	aggregated data observations on address a

```

1: function CALCPROBABILITY( $B$ )
2:    $G = \text{BUILDUCFG}(B)$ 
3:    $O_{code}, O_{data} = \text{COLLECTOBSERVATIONS}(B)$ 
4:    $P = \text{ONESTEPSUMPRODUCT}(G, O_{code}, O_{data})$ 
5:   return  $P$ 
6: end function
7:
8: function ONESTEPSUMPRODUCT( $G, O_{code}, O_{data}$ )
9:    $G_{DAG} = \text{TRANSFROMINTODAG}(G)$   $\triangleright$  Transform  $G$  into a Directed Acyclic Graph (DAG) via
   collapsing each Strongly Connected Component (SCC) into a vertex
10:   $O_{DAG\_code} = \text{CREATEEMPTYMAPPING}()$   $\triangleright$  DAG-related mapping, initialized as empty
11:
12:                                      $\triangleright$  Step 1: Aggregate code observation values
13:  for each SCC  $x$  in topological order of  $G_{DAG}$  do
14:     $o_1 = \text{PRODUCT}(\{O_{DAG\_code}[y] \mid \text{SCC } y \text{ is a predecessor of SCC } x\})$ 
15:     $o_2 = \text{PRODUCT}(\{O_{code}[i] \mid \text{address } i \text{ belongs to SCC } x\})$ 
16:     $O_{DAG\_code}[x] = o_1 \times o_2$ 
17:    for each address  $i$  in all addresses belonging to SCC  $x$  do
18:       $O_{code}[i] = O_{DAG\_code}[x]$ 
19:    end for
20:  end for
21:                                      $\triangleright$  Step 2: Aggregate data observation values
22:                                      $\triangleright$  The last address whose  $O_{data} > 0$ 
23:   $i_{prev} = \infty$ 
24:  for each address  $i$  of  $B$  in increasing order do
25:    if  $O_{data}[i] \neq \perp$  then
26:      if  $1 < i - i_{prev} < D$  then
27:         $o_1, o_2 = O_{data}[i], O_{data}[i_{prev}]$ 
28:        for each address  $j \in (i_{prev}, i)$  do
29:           $O_{data}[j] = 1 - \frac{p_{data}}{2p_{data} + (1-o_1)(1-o_2) - 2p_{data}(1-o_1)(1-o_2)}$ 
30:        end for
31:      end if
32:       $i_{prev} = i$ 
33:    end if
34:  end for
35:                                      $\triangleright$  Step 3: one-step sum-product for each address
36:  for each address  $i$  of  $B$  in increasing order do
37:     $o_{neg} = (O_{code}[i] = \perp ? 0.5 : O_{code}[i])$ 
38:     $o_{pos} = (O_{data}[i] = \perp ? 0.5 : O_{data}[i])$ 
39:     $P[i] = o_{neg} \cdot (1 - o_{pos}) / (o_{pos} \cdot (1 - o_{neg}) + o_{neg} \cdot (1 - o_{pos}))$ 
40:  end for
41:  return  $P$ 
42: end function

```



$$\begin{aligned}
P(a = 1) &= \frac{\mathcal{F}_{code}(1) \cdot \mathcal{F}_{data}(1)}{\mathcal{F}_{code}(1) \cdot \mathcal{F}_{data}(1) + \mathcal{F}_{code}(0) \cdot \mathcal{F}_{data}(0)} \\
&= \frac{o_{neg} \cdot (1 - o_{pos})}{o_{neg} \cdot (1 - o_{pos}) + o_{pos} \cdot (1 - o_{neg})}
\end{aligned}$$

Figure 4.9. Factor Graph for Each Address

tation flexibility) and retains data in the original space. And the original code is replaced with `hlt`. Its rewriting ensures a critical property: *a rewritten instruction should evaluate to the same value(s) as its original version*. This ensures all data accesses (to the original space) are not broken. For example, a rewritten read of `rip` must be patched with an offset such that the read yields the corresponding value in the original space as the rewritten read must be executed in the shadow space.

Specifically, it performs the following code transformations. It directly patches direct jump instructions by an offset statically computed based on the offset between the shadow and original address spaces and the instrumentations. The computation of such offset is standard and elided [38]. It instruments all indirect jumps to perform a runtime address lookup that translates the target to the shadow space. It may throw an intentional segfault if it detects the target is not in the shadow space, meaning the corresponding code has not been rewritten. Client analysis instrumentation such as coverage tracking code is inserted in the shadow space.

Handling Call Instructions to Support Data Accesses through Return Addresses.

There are programs that access data using addresses computed from some return address on the stack. As such, we need to ensure return addresses saved on the stack must be those in the original space. Therefore, STOCHFuzz rewrites a call instruction to a `push`

instruction which pushes a patched return address (pointing to the original address) to the stack, followed by a `jmp` instruction to the callee in the shadow space. We then instrument `ret` instructions to conduct on-the-fly lookup just like in handling indirect jumps.

Our design allows keeping the control flow in the shadow space as much as possible, which can improve instruction cache performance. An exception is callbacks from external libraries, which cause control flow to the original space, even though it quickly jumps back to the shadow space.

Generating Random Binary Versions. Besides the aforementioned transformations, STOCHFuzz also performs the following stochastic rewriting to generate a pool of N different binaries (every time the rewriter is invoked). Specifically, for addresses whose their probabilities of being data are smaller than a threshold p_θ but not 0 (i.e., not “certainly code” but “likely code”), they have a chance of $1 - p_\theta$ to be replaced with `hlt`. In our setting, we have $N = 10$ and $p_\theta = 0.01$.

4.3.3 Crash Analyzer

Recall that the crash analyzer needs to decide if a crash is due to a rewriting error. If so, it needs to locate and repair the crash inducing rewriting error. Let S be a set of uncertain addresses (that *may* be replaced with `hlt`), and $\mathcal{R}(S)$ the execution result of a rewritten binary where all the addresses in S are replaced with `hlt`. Assume $\mathcal{R}(S_1)$ yields an unintentional crash. To determine whether the crash is caused by a rewriting error, the analyzer compares the results of $\mathcal{R}(S_1)$ and $\mathcal{R}(\emptyset)$. If $\mathcal{R}(S_1) = \mathcal{R}(\emptyset)$, the crash is caused by a latent bug in the subject program, and vice versa.

Then, locating the crash inducing rewriting error can be formalized as finding a *1-minimal* subset $S_2 \subseteq S_1$, which satisfies $\mathcal{R}(S_2) = \mathcal{R}(S_1)$ and $\forall a_i \in S_2 : \mathcal{R}(S_2 \setminus \{a_i\}) \neq \mathcal{R}(S_1)$ [102]. Intuitively, all the addresses in S_2 must be erroneously replaced with `hlt`. It can be proved by contradiction. Assuming $a_j \in S_2$ is a code byte (and hence its rewriting is correct), not replacing address a_j (with `hlt`) should not influence the execution result, that is $\mathcal{R}(S_2 \setminus \{a_j\}) = \mathcal{R}(S_2)$. As $\mathcal{R}(S_2) = \mathcal{R}(S_1)$, $\mathcal{R}(S_2 \setminus \{a_j\}) = \mathcal{R}(S_1)$, directly contradicting with the 1-

Algorithm 8 Register Liveness Analysis on UCFG

INPUT:	B	binary indexed by address
OUTPUT:	$D[i] \subseteq \{r_1, r_2, \dots\}$	dead registers at address i

```

1: function ANALYZEDEADREG( $B$ )
2:    $D = \text{CREATEEMPTYMAPPING}()$ 
3:   for each address  $i$  of  $B$  in decreasing order do
4:      $Succ = \{j \mid \text{ExplicitSucc}(i, j)\}$   $\triangleright Succ = \emptyset$  if  $i$  is an indirect jump/call
5:     if  $\exists j \in Succ$ , s.t.  $j \leq i$  then
6:        $d_{\text{after}} = \{\}$   $\triangleright$  Assume there is no dead variable after executing address  $i$ 
7:     else
8:        $d_{\text{after}} = \bigcap_{j \in Succ} D[j]$ 
9:     end if
10:     $D[i] = (d_{\text{after}} \cup \{r_w \mid \text{RegWrite}(i, r_w)\}) \setminus \{r_r \mid \text{RegRead}(i, r_r)\}$ 
11:  end for
12: end function

```

minimal property. Delta debugging [102] is an efficient debugging technique that guarantees to find 1-minimal errors. It operates in a way similar to binary search. Details are elided.

4.3.4 Optimizations

We develop three optimizations for STOCHFuzz, which are directly performed on rewritten binaries without lifting to IR.

Register Reuse. Instrumentation may need to use registers. To avoid breaking program semantics, inside each instrumentation code block, registers need to be saved at the beginning and restored at the end. These context savings become performance bottleneck. We perform a register liveness analysis such that *dead registers*, which hold some value that will never be used in the future, can be reused in instrumentation. The difference between our liveness analysis and a traditional liveness analysis is that ours is performed on the UCFG.

Algorithm 8 presents the analysis. It takes a binary and outputs a mapping from an address i to a set of registers which are dead at i . The algorithm traverses all addresses in a descendent order (line 3). For each address i , the algorithm first collects the explicit successors of i in UCFG (line 4). If there is at least one successor whose address is smaller than i , which indicates the successor has not been analyzed (line 5), the algorithm conser-

vatively assumes all the registers are not dead *after* i (line 6). Otherwise, the registers that are dead *at* all successors are marked as dead *after* i (line 8). At last, the dead registers *at* i are computed from the dead registers *after* i and the i instruction itself (line 10). Specifically, the registers written by i become dead (as the original values in those registers are no longer used beyond i); the ones read by i are marked live and removed from the dead set as i needs their values. Upon instrumentation, STOCHFuzz reuses the dead registers at the instrumentation point.

Removing Flag Register Savings. Saving and restoring flag registers has around $10\times$ more overhead compared with general purpose registers. We perform the same register liveness analysis on flag registers and avoid saving/restoring the dead ones.

Removing Redundant Instrumentation. If a basic block has only one successor, its successor is guaranteed to be covered once the block is covered [106]. We hence avoid instrumenting these single successors.

4.4 Probabilistic Guarantees

In this section, we study the probabilistic guarantees of STOCHFuzz. We focus on two aspects. The first is the likelihood of rewriting errors (i.e., data bytes are mistakenly replaced with `hlt`) corrupting coverage information without triggering a crash. Note that if it triggers a crash, STOCHFuzz can locate and repair the error. The second is the likelihood of instruction bytes not being replaced with `hlt` so that we miss coverage information. Note there is no crash in this case but rather some instructions are invisible to our system and not rewritten. Our theoretical analysis shows that the former likelihood is 0.05% and the latter is 0.01% (with a number of conservative assumptions). They are also validated by our experiments.

Likelihood of Rewriting Error Not Causing Crash But Corrupting Coverage Feedback. If the rewriting error does not change execution path, it does not corrupt coverage feedback. In this case, we are not worried about the rewriting error even if it does not cause a crash. In other words, we are only interested in knowing the likelihood of a rewriting error changes program path but does not induce crash *over all the fuzzing runs*.

Note that as long as it causes crash in one fuzzing run, STOCHFuzz can catch and repair it. This is the strength of having a stochastic solution. In our study, we use the following definitions.

- M : the number of fuzzing executions
- p_{fp} : the likelihood that a data byte is classified as code and subject to replacement (with `hlt`), we call it a false positive (FP).
- $p_{\text{patch}} = 1 - p_{\theta}$: how likely a code byte (classified by STOCHFuzz) is selected for replacement in a rewritten binary.
- p_{crash} : the likelihood that a mistakenly replaced data byte changes program path and crashes *in a single execution*.

From the above definitions, the likelihood of a data byte is mistakenly patched is $p_{\text{fp}} \times p_{\text{patch}}$. The likelihood of a data byte being patched and triggering a crash (hence STOCHFuzz observes and repairs it) is $p_{\text{fp}} \times p_{\text{patch}} \times p_{\text{crash}}$.

The likelihood of the error escapes STOCHFuzz in M executions is hence the following.

$$(1 - p_{\text{fp}} \times p_{\text{patch}} \times p_{\text{crash}})^M$$

With a conservative setting of $p_{\text{fp}} = 0.015$, the average initial FP rate according to our experiment (Section 4.6.2, $p_{\text{patch}} = 0.99$, $p_{\text{crash}} = 0.0005$ (a very conservative setting as in practice it is over 90%), and $M = 1,000,000$, STOCHFuzz has 0.05% chance missing the error. We want to point out that if p_{crash} is 0, meaning the error always changes path without crashing, STOCHFuzz can never detect it. We haven't seen such cases in practice. One way to mitigate the issue is to use other instructions similar to `hlt` in patching.

Likelihood of Missing Coverage Due to Code Bytes Not Being Patched. Intuitively, the likelihood is low for two reasons. First, coverage information is collected at the basic block level. Missing coverage only happens when STOCHFuzz mis-classifies all the code bytes in a basic block to data. Second, even if STOCHFuzz considers a code byte is likely

data, there is still a chance it is chosen for patching during stochastic rewriting. Over a large number of fuzzing runs, STOCHFuzz can expose it through an intentional crash.

To simplify our discussion, we only consider the second reasoning. In other words, we consider missing coverage at the byte level (not basic block level). We use the following definitions in addition to the previous ones.

- p_{fn} : the likelihood STOCHFuzz mis-classifies a code byte to data, called a false negative (FN).
- p_{exe} : the likelihood a code byte is covered in an execution.

The likelihood of a code byte being chosen for patching in a binary version is $(1 - p_{\text{fn}}) \times p_{\text{patch}}$. The likelihood of a code byte being patched and covered in an execution (and hence STOCHFuzz detects it) is $(1 - p_{\text{fn}}) \times p_{\text{patch}} \times p_{\text{exe}}$.

The likelihood that the rewriting error escapes from STOCHFuzz in M runs is hence the following.

$$(1 - (1 - p_{\text{fn}}) \times p_{\text{patch}} \times p_{\text{exe}})^M$$

With a practical setting of $p_{\text{fn}} = 0.12$ (the average initial FN rate of STOCHFuzz according to our experiment), $p_{\text{patch}} = 0.99$, $p_{\text{exe}} = 1e-5$ (a very conservative setting), $M = 1,000,000$, STOCHFuzz has 0.01% chance missing the error. We want to point out that if p_{exe} is 0, meaning the code byte is never executed in any runs, STOCHFuzz can never detect it. However, in such cases, the error has no effect on fuzzing and hence unimportant. Also note that if we consider coverage at basic block level, the bound can be lower.

4.5 Practical Challenges

We have addressed a number of practical challenges such as supporting exception handling in C++, reducing process set up cost, safeguarding non-crashing rewriting errors, and handling occluded rewriting.

Supporting Exception Handling in C++. Exception handling in C++ poses additional challenges for static rewriting [78]. Specifically, when handling exceptions, the program needs to acquire the return addresses pushed by previous call instructions to unwind stack frames.

To support this, `STOCHFuzz` additionally intercepts calls to external library functions and replace their return addresses (in the shadow space) with the corresponding addresses in the original space. Note that this is different from our transformation of call instructions to a push followed by a jump. As such, when execution returns from external libraries, it goes to the original space instead of the shadow space, incurring additional control flow transfers. To reduce the overhead, a white-list of widely-used library functions, for which we do not need to intercept the calls, is used. We argue it is a one-time effort and can be done even for closed-source programs, as the symbols of external library functions are always exposed. To understand the worst-case performance of `STOCHFuzz`, we disable the white-list optimization during evaluation.

Efficient Process Set Up. Setting up a process (e.g., linking and library initialization) has a relatively high overhead. To avoid it, a fork server, which communicates with the fuzzer through Linux pipe and forks the subject process once requested, is instrumented into the subject program by AFL. In `STOCHFuzz`, the dispatcher is a component of AFL, which sets up N fork servers prior to fuzzing and randomly selects one to communicate with when requesting an execution instance. Additionally, for each rewritten binary, its original and shadow spaces are both re-mapped as shared memory with the incremental rewriter. As such, during fuzzing, the incremental and stochastic rewriting does not trigger any process set up cost.

Safeguarding Non-crashing Rewriting Errors. During fuzzing, AFL automatically monitors an input *stability* metric which measures the consistency of observed traces [107]. That is, if the subject program always behaves the same for the same input data, the fuzzing stability earns a score of 100%. A low score suggests low input consistency. This metric can help `STOCHFuzz` detect rewriting error which does not trigger a crash but changes execution trace. Specifically, once this metric becomes smaller than a given threshold, the rewriting error localization procedure is triggered. As such, the soundness guarantee of `STOCHFuzz` can be stronger than the one calculated in Section 4.4 in practice. In our evaluation, we turn it off to measure worst-case performance.

Handling Occluded Rewriting. Another practical challenge is to handle the case in which `hlt` is mistakenly placed inside a true instruction (e.g., replacing address 1 inside the true `mov` instruction at 0 in Figure 4.6). As such, the address which triggers a crash may not be the address of the inserted `hlt`. Although it is highly unlikely in practice, our crash analyzer could not repair the error properly when it happens. To handle the problem, we design a set of advanced rewriting rules, which guarantees control flow will be terminated at a set of pre-selected addresses once an occluded instruction gets executed. As such, we can infer there is an occluded rewriting error. Specifically, for a given address a with $Inst(a, c)$, we use the following rules to rewrite it:

1. Check whether a is occluded with any control flow transfer instruction (starting at an earlier address). If so, avoid replacing it;
2. Replace all addresses between a and a_m where $a_m = \max(\{a_i + c_i \mid Inst(a_i, c_i) \wedge (a_i < a < a_i + c_i)\})$, meaning the maximum end address of an instruction occluded with a .

As such, any execution that encounters an instruction occluded with some injected `hlt` must be terminated at an address in $S_c = \{a_i \mid Inst(a_i, c_i) \wedge (a_i < a < a_i + c_i)\} \cup [a, a_m]$.

Proof. Assume an execution goes through the patched addresses $[a, a_m]$ and $Inst(a_c, c_c)$ is the first instruction corrupted by some of the patches. We hence have $[a_c, c_c) \cap [a, a_m] \neq \emptyset$. There are three cases.

- If $a_c < a$ and the corrupted instruction causes a crash, the address reported to our crash analyzer is a_c . Since $[a_c, c_c) \cap [a, a_m] \neq \emptyset$ and $a_c < a$, we have $a_c < a < a_c + c_c$. Hence, $a_c \in \{a_i \mid Inst(a_i, c_i) \wedge (a_i < a < a_i + c_i)\} \subseteq S_c$;
- If $a_c < a$ and the corrupted instruction does not cause a crash, the next executed instruction starts at $a_c + c_c$ because rule 1 guarantees $Inst(a_c, c_c)$ is not a control flow transfer instruction. Similarly, as $[a_c, c_c) \cap [a, a_m] \neq \emptyset$ and $a_c < a$, we have $a_c < a < a_c + c_c$. According to the definition of a_m , $a_c + c_c \leq a_m$. So the executed instruction is a `hlt` patched at address $a_c + c_c \in [a, a_m] \subseteq S_c$.

Table 4.2. Soundness on Google FTS (**X** means failure)

Program	<i>afl-qemu</i>	<i>ptfuzzer</i>	<i>e9patch</i>	<i>ddisasm</i>	STOCHFuzz
boringssl			X	X	✓
freetype2		X		X	✓
guetzli		X			✓
harfbuzz				X	✓
lcms				X	✓
libarchive		X			✓
libxml2	X			X	✓
openssl-1.0.1f			X	X	✓
openssl-1.0.2d			X		✓
openssl-1.1.0c			X	X	✓
openthread				X	✓
sqlite		X			✓
wpantund				X	✓

- If $a \leq a_c \leq a_m$, it happens when a_c is a jump/call target. As all addresses in $[a, a_m]$ are patched, it will directly crash by a `hlt` instruction. Hence, the crashed address is a_c itself where $a_c \in [a, a_m] \subset S_c$.

Note that a_c cannot be larger than a_m , according to $[a_c, c_c) \cap [a, a_m] \neq \emptyset$. \square

4.6 Evaluation

STOCHFuzz is implemented from scratch with over 10,000 lines of C code, leveraging Capstone [108] and Keystone [109] that provide basic disassembling and assembling functionalities, respectively. Our evaluation takes more than 5000 CPU hours and is conducted on three benchmark sets, including the Google Fuzzer Test Suite (Google FTS) [91], a variant of Google FTS which is compiled with inlined data, and the fuzzing benchmarks from *RetroWrite* [78]. We compare STOCHFuzz with the state-of-the-art binary-only fuzzers, including *ptfuzzer*, *afl-qemu*, *RetroWrite*, *e9patch*, and *ddisasm*. In addition, we use STOCHFuzz on 7 commercial binaries and find 2 zero-days. We port a recent work IJON [95] on state-based fuzzing to support stripped binaries, demonstrating STOCHFuzz can collect other feedback than coverage.

All the benchmarks are compiled by Clang 6.0 with their default compilation flags (“-O2” in most cases). For *e9patch*, as it cannot recover CFG from a stripped binary, we instrument all the control flow transfer instructions (e.g., `jmp`) to trace the execution paths. For *ddisasm*, the version we use is 1.0.1, and the reassembly flags we use are “-no-cfi-directives” and “-asm”. The reassembly of *ddisasm* is performed on a server equipped with a 48-cores CPU (Intel(R) Xeon(R) Silver 4214 CPU @ 2.20GHz) and 188G main memory. All others are conducted on a server equipped with a 12-cores CPU (Intel(R) Core(TM) i7-8700 CPU @ 3.20GHz) and 16G main memory.

4.6.1 Evaluation on Google FTS

Google FTS is a standard benchmark widely used to evaluate fuzzing techniques [82, 110, 111], consisting of 24 complex real-world programs. We compare STOCHFuzz with *ptfuzzer*, *afl-qemu*, *e9patch*, and *ddisasm*. We additionally compare with two compiler-based baselines (*afl-gcc* and *afl-clang-fast*). However, we cannot compare with *RetroWrite* on Google FTS as *RetroWrite* cannot instrument stripped binaries and it requires the binaries not written in C++, while all the binaries are stripped in this experiment and 1/3 of them are C++ ones.

Soundness. Table 4.2 presents the overall soundness of binary-only fuzzing solutions. The first column shows the programs. Columns 2-6 show whether *afl-qemu*, *ptfuzzer*, *e9patch*, *ddisasm*, and STOCHFuzz successfully generate binaries that the fuzzer can execute, respectively. Note that we only present the programs which at least one tool fails to instrument (due to the space limitations). Specifically, *afl-qemu* fails on *libxml2* due to a known implementation bug [112], *ptfuzzer* fails on 4 out of the 24 programs due to unsolved issues in their implementation [113], *e9patch* fails on 4 programs as these programs contain hand-written assembly code interleaved with data, *ddisasm* fails on 9 programs which crash on the seed inputs after reassembly due to uncertainty in their heuristics¹, and STOCHFuzz succeeds on all the 24 programs.

Fuzzing Efficiency. To assess the fuzzing efficiency achieved by STOCHFuzz, we run AFL to fuzz the instrumented binaries for 24 hours. Figure 4.10 presents the total number of

¹After being reported to the developers of *ddisasm*, 6 out of 9 test failures got fixed in the latest release (via strengthening heuristics). Details can be found at <https://github.com/GrammaTech/ddisasm/issues/20>.

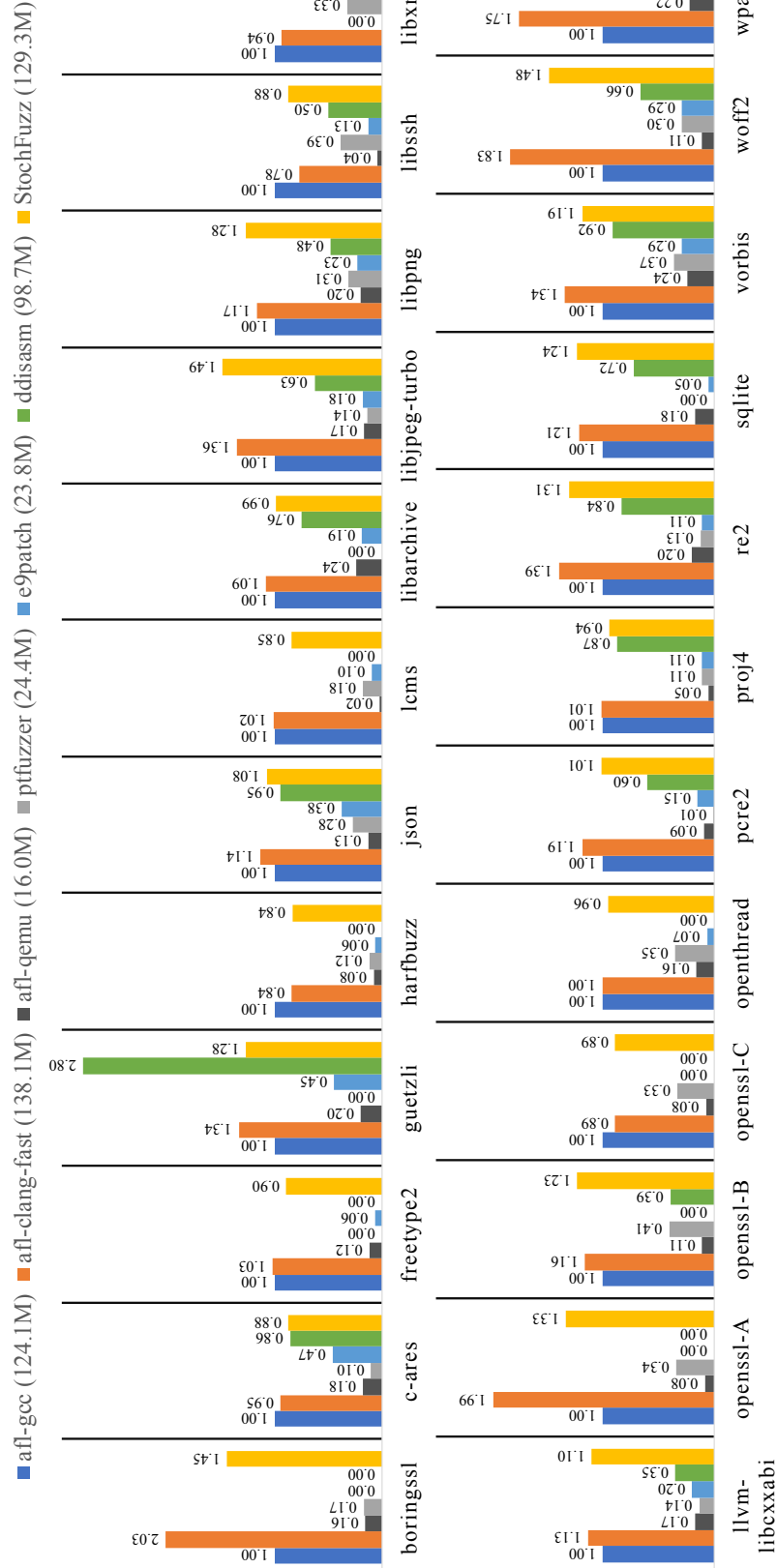


Figure 4.10. Total number of fuzzing executions of each tool in 24 hours

Table 4.3. Mean and standard deviation of time-to-discovery (in minutes) for bugs in Google FTS

Tool	guetzli	json	llvm-libcxxabi
<i>afl-gcc</i>	513.25 ± 114.84	0.85 ± 0.63	0.08 ± 0.00
<i>afl-clang-fast</i>	539.56 ± 240.83	0.18 ± 0.17	0.08 ± 0.00
<i>afl-qemu</i>	$+\infty$	2.64 ± 3.56	0.23 ± 0.05
<i>ptfuzzer</i>	$+\infty$	49.08 ± 82.35	0.79 ± 0.25
<i>e9patch</i>	$+\infty$	21.87 ± 36.21	0.35 ± 0.00
<i>ddisasm</i>	505.22 ± 93.45	N/A	0.08 ± 0.00
STOCHFuzz	363.37 ± 120.14	0.67 ± 1.02	0.08 ± 0.00

Tool	pcre2	re2	woff2
<i>afl-gcc</i>	763.61 ± 40.44	2.21 ± 2.14	12.89 ± 0.44
<i>afl-clang-fast</i>	461.73 ± 219.89	3.08 ± 3.93	12.09 ± 4.91
<i>afl-qemu</i>	$+\infty$	$+\infty$	67.23 ± 26.94
<i>ptfuzzer</i>	$+\infty$	42.92 ± 68.08	29.18 ± 0.19
<i>e9patch</i>	$+\infty$	$+\infty$	30.73 ± 0.28
<i>ddisasm</i>	913.90 ± 495.42	N/A	14.60 ± 0.25
STOCHFuzz	768.91 ± 264.82	2.32 ± 0.54	7.43 ± 0.27

fuzzing executions, where we take *afl-gcc* as a baseline and report the ratio of each tool to *afl-gcc*. Larger numbers indicate better performance. The average numbers of fuzzing executions over the 24 programs are presented in the legend (on the top) associated with the tools. STOCHFuzz outperforms *afl-gcc* in 13 out of 24 programs. For the remaining 11 programs, STOCHFuzz also achieves comparable performance with *afl-gcc*. *Afl-clang-fast* achieves the best performance among all the tools, as it does instrumentation at the IR level. Compared with it, STOCHFuzz has 11.77% slowdown on average due to the additional overhead of extra control flow transfers (from the original space to the shadow space) and switching between binary versions. *Ddisasm* also achieves good performance. However, due to its inherent soundness issues, it fails on 9 out of the 24 programs. Other tools have relatively higher overhead.

Bug Finding. As Time-to-discovery (TTD) (of bugs) directly reflects fuzzing effectiveness, and hence suggests instrumentation effectiveness and fuzzing throughput, we additionally conduct an experiment to show the time needed to find the first bug for each tool. We run

each tool three times with a 24-hour timeout. Table 4.3 shows the average TTD (in minutes) and the standard deviation. We only report the programs for which at least one tool can report a bug within the time bound. The first column presents the tools. Columns 2-4 show the TTDs for different programs. The symbol $+\infty$ denotes the tool cannot discover any bug within the time bound. *N/A* denotes the crash(es)² discovered by the tool cannot be reproduced by executing the non-instrumented binary. Due to their high overhead, *afl-qemu*, *ptfuzzer*, and *e9patch* cannot discover bugs in multiple programs. Although *ddisasm* achieves good performance in the programs that it can instrument, it generates invalid crashes for some programs due to its soundness issues. STOCHFuzz has a similar TTD to *afl-gcc*. This shows the soundness and effectiveness of STOCHFuzz.

We also collect the path coverage in 24 hours. The average coverage for *afl-gcc*, *afl-clang-fast*, and STOCHFuzz is 2572, 2239, and 2493, respectively. As other tools do not work on all the programs, their numbers are not comparable, and hence elided. We also omit the details due to the page limitations.

Optimization Effectiveness. Table 4.4 presents the effects of optimizations. The second column presents the number of executed blocks during fuzzing. Columns 3-4, 5-6, and 7-8 present the results for removing flag register savings (FLAG), general purpose register reuse (GPR), and removing instrumentation for single successors, respectively. For each optimization, we report both the number (of applying these optimizations) and the percentage. In the last column, we present the slow-down when the optimizations are disabled. Overall, FLAG is most effective, removing 99% of cases. Intuitively, the use of flag registers has very strong locality. We then conduct a study on the evaluated binaries and find that almost all flag registers are defined and used within the last three instructions of basic blocks, with the most common instruction pattern being a `cmp` or `test` instruction followed by a conditional jump. As such, they are mostly dead at the instrumentation points. GPR can be applied in 82.2% cases on average. The observation is that many basic blocks start with instructions that write to at least one general purpose register. STOCHFuzz hence is able to reuse the register in the instrumented code (Section 4.3.4). The average percentage of instrumentation

²↑The latest *ddisasm* can correctly reassemble all N/A programs.

Table 4.4. Effects of optimizations. #B denotes the number of basic blocks instrumented by STOCHFuzz, #O denotes the number of blocks where an optimization is applied at least once, %R denotes the percentage, and %S denotes the slowdown when disabling the optimizations.

Program	#B	FLAG		GPR		Single-Succ		%S
		#O	%R	#O	%R	#O	%R	
boringsssl	5,112	5,068	99.1	4,294	84.0	2,225	43.53	37.81
c-ares	98	96	98.0	83	84.7	48	48.98	-4.22
freetype2	13,590	13,508	99.4	11,422	84.0	6,126	45.08	1.86
guetzli	10,680	10,621	99.4	8,230	77.1	5,312	49.74	3.49
harfbuzz	10,365	10,208	98.5	8,679	83.7	4,256	41.06	28.39
json	2,308	2,296	99.5	1,886	81.7	1,125	48.74	30.48
lcms	4,256	4,181	98.2	3,341	78.5	1,712	40.23	-16.58
libarchive	7,134	7,046	98.8	5,862	82.2	2,540	35.60	33.25
libjpeg-turbo	2,953	2,927	99.1	2,609	88.4	1,362	46.12	35.48
libpng	2,815	2,797	99.4	2,173	77.2	1,153	40.96	18.07
libssh	4,441	4,393	98.9	3,578	80.6	1,816	40.89	30.43
libxml2	13,546	13,487	99.6	10,786	79.6	5,531	40.83	15.96
llvm-libcxxabi	4,257	4,244	99.7	3,314	77.8	2,171	51.00	28.77
openssl-1.0.1f	15,912	15,750	99.0	13,595	85.4	7,028	44.17	43.88
openssl-1.0.2d	2,347	2,285	97.4	2,036	86.7	961	40.95	64.24
openssl-1.1.0c	6,964	6,902	99.1	5,856	84.1	1,970	42.66	16.03
openthread	6,074	6,048	99.6	4,878	80.3	2,387	39.30	14.27
pcre2	6,889	6,798	98.7	5,863	85.1	3,292	47.79	45.86
proj4	1,983	1,915	96.6	1,443	72.8	984	49.62	3.58
re2	6,693	6,655	99.4	5,140	76.8	3,382	50.53	31.05
sqlite	24,264	24,128	99.4	20,541	84.7	11,314	46.63	38.87
vorbis	3,297	3,263	99.0	2,539	77.0	1,375	41.70	16.95
woff2	2,406	2,374	98.7	1,990	82.7	1,191	49.50	30.92
wpantund	27,549	27,146	98.5	22,765	82.6	11,587	42.06	2.10
Average	7,747	7,672	99.0	6,371	82.2	3,410	44.49	22.45

removal for blocks with a single successor is 44.49%, which is not that significant but still helpful. The slowdown is 22.45% on average when we disable these optimizations. The optimizations have negative effects on some programs such as *lcms*. Further inspection seems to indicate that the optimizations cause some tricky complications in cache performance. It is worth pointing out that compiler based fuzzers such as *afl-gcc* and *afl-clang* directly benefit from built-in compiler optimizations, some of which have similar nature to ours. Dynamic instrumentation engines such as QEMU and PIN have their own optimizations although they typically reallocate all registers. Performing optimizations during unsound static rewriting

Table 4.5. Analysis and rewriting overhead

Program	<i>e9patch</i>	<i>ddisasm</i>	<i>ddisasm</i>	STOCHFuzz	
		default (-j48)	-j8	rewriting	prob. anly.
boringssl	-	67h 43m 20s	126.90s	9.77s	67.35s
c-ares	0.02s	0h 47m 22s	1.17s	0.05s	0.02s
freetype2	0.76s	28h 57m 28s	96.24s	21.39s	91.59s
guetzli	0.38s	8h 51m 19s	76.05s	5.47s	95.84s
harfbuzz	0.51s	8h 02m 28s	70.89s	5.33s	64.17s
json	0.10s	4h 44m 48s	12.93s	1.30s	8.33s
lcms	0.34s	10h 39m 50s	36.58s	3.56s	13.19s
libarchive	0.51s	11h 53m 49s	61.67s	4.09s	34.29
libjpeg-turbo	0.45s	30h 16m 04s	108.79s	10.49s	24.33s
libpng	0.13s	3h 29m 24s	10.87s	1.48s	3.54s
libssh	0.36s	54h 03m 58s	50.22s	2.74s	23.98s
libxml2	2.03s	23h 52m 25s	188.59s	19.86s	177.20s
llvm-libcxxabi	0.19s	4h 33m 28s	15.57s	1.90s	19.78s
openssl-1.0.1f	-	83h 57m 03s	209.57s	22.95s	153.62s
openssl-1.0.2d	-	25h 05m 28s	37.91s	2.55s	4.82s
openssl-1.1.0c	-	117h 15m 42s	354.86s	31.57s	229.91s
openthread	0.70s	20h 24m 43s	57.96s	6.10s	13.33s
pcre2	0.33s	26h 35m 04s	481.04s	4.38s	24.38s
proj4	0.42s	10h 43m 34s	39.25s	4.69s	20.62s
re2	0.40s	17h 12m 33s	41.62s	4.60s	84.82s
sqlite	1.02s	16h 49m 43s	117.92s	14.38s	233.97s
vorbis	0.22s	16h 07m 57s	32.29s	2.26s	12.61s
woff2	0.49s	39h 09m 27s	123.50s	6.34s	21.11s
wpantund	1.58s	33h 08m 02s	176.65s	14.55s	579.94s
Average	0.55s	27h 41m 02s	105.38s	8.41s	83.41s

is very risky. In contrast, optimizations work well in our context as STOCHFuzz can fix disassembly and rewriting errors automatically.

Analysis and Rewriting Overhead on Google FTS. Different from techniques leveraging hardware features or dynamic translation, techniques based on static rewriting incur analysis and rewriting cost. We further study such overhead on the standard Google FTS for *e9patch*, *ddisasm*, and STOCHFuzz. Table 4.5 shows the results (measured by total CPU time). The second column shows the overhead of *e9patch*. The third and fourth columns show the overhead of *ddisasm* using different reassembly flags, and the last two columns show the overhead of STOCHFuzz which is broken down to rewriting and probability analysis overhead. Note that *ddisasm* uses all 48 cores by default. However, after communicating

with the developers, we were notified that there are some parallelism issues with the default setting. As such, running with `-j8` (for using 8 cores) produces much better results. *E9patch* does not distinguish code and data, as it assumes exclusion of such interleavings. Hence, it has the lowest cost. Although the aggregated overheads of STOCHFuzz are not trivial, they are amortized over the 24 hours period. Also observe that STOCHFuzz’s overhead is comparable to *ddisasm* (`-j8`).

4.6.2 Evaluation on Google FTS with Intentional Data Inlining

Programs built by popular compilers (e.g., GCC and Clang) with default settings may not contain (substantial) code and data interleavings [70]. It is interesting to study the performance of various tools when substantial interleavings are present. We hence modify the compilation tool-chain of Google FTS to force `.rodata` sections to be interleaved with `.text` sections. We extract the ground-truth of data byte locations from the debugging information and then strip the binaries.

Table 4.6 presents the overall effectiveness results for the experiment on Google FTS with intentional data inlining. The numbers of inlined data bytes are presented in the second column (i.e., data bytes in between two code sections), and whether the binaries instrumented by *e9patch*, *ddisasm*, and STOCHFuzz can be successfully fuzzed are presented in the next three columns, respectively. *E9patch* fails on 22 out of the 24 programs, due to its assumption of no inlined data. It succeeds on two programs because they do not contain static data sections. *Ddisasm* fails on 21 programs due to three reasons. Specifically, \mathbf{X}_1 denotes a recompilation error that a byte value is larger than 256. It happens when *ddisasm* mis-classifies a data byte as an offset of two labels. Hence, when instrumentation code is inserted, the offset increases, making the data byte larger than 256. Symbol \mathbf{X}_2 denotes a recompilation error that the target of a jump instruction is an integer (instead of a symbol). It happens when *ddisasm* mis-classifies some data bytes as a jump instruction whose target cannot be symbolized. Symbol \mathbf{X}_3 denotes that instrumentation code crashes on seed inputs (due to some recompilation error). In contrast, STOCHFuzz successfully instruments and fuzzes all the programs.

Table 4.6. Effectiveness on Google FTS w/ Intentional Data Inlining

Program	# Inlined Data Bytes	<i>e9patch</i>	<i>ddisasm</i>	STOCHFuzz
boringssl	263,539	✗	✗ ₃	✓
c-ares	7	✗		✓
freetype2	91,960	✗	✗ ₂	✓
guetzli	18,543	✗	✗ ₃	✓
harfbuzz	63,061	✗	✗ ₃	✓
json	0			✓
lcms	22,576	✗	✗ ₂	✓
libarchive	55,698	✗	✗ ₃	✓
libjpeg-turbo	79,329	✗	✗ ₃	✓
libpng	9,054	✗	✗ ₂	✓
libssh	141,943	✗	✗ ₃	✓
libxml2	128,007	✗	✗ ₃	✓
llvm-libcxxabi	0			✓
openssl-1.0.1f	169,787	✗	✗ ₃	✓
openssl-1.0.2d	43,796	✗	✗ ₂	✓
openssl-1.1.0c	369,397	✗	✗ ₂	✓
openthread	32,691	✗	✗ ₃	✓
pcre2	95,763	✗	✗ ₁	✓
proj4	30,978	✗	✗ ₂	✓
re2	35,336	✗	✗ ₂	✓
sqlite	35,467	✗	✗ ₃	✓
vorbis	59,986	✗	✗ ₂	✓
woff2	494,994	✗	✗ ₂	✓
wpantund	89,203	✗	✗ ₂	✓

Fuzzing Efficiency. We run the tools for 24 hours on each program. Figure 4.11 presents the number of fuzzing executions by our tool and its ratio over *afl-gcc*. We omit the results for other tools as inlined data do not impact their efficiency in theory. The results show that STOCHFuzz still has comparable performance as *afl-gcc*. Moreover, our tool’s efficiency has a slight degradation compared to without intentional data inlining (124.7M v/s 129.3M), due to the extra time needed to fix more rewriting errors.

Progress of Incremental and Stochastic Rewriting. We study how the numbers of false positives (FPs) (i.e., a data byte is replaced with `hlt`) and false negatives (FNs) (i.e., a code byte is not replaced with `hlt`) change over the procedure. Here, we use debugging information and the aggregated coverage information (over 24-hour fuzzing) to extract the ground-truth. In other words, we do not consider data bytes that are not accessed in the 24 hours and code bytes that are not covered in the 24 hours. Note that they have no influence

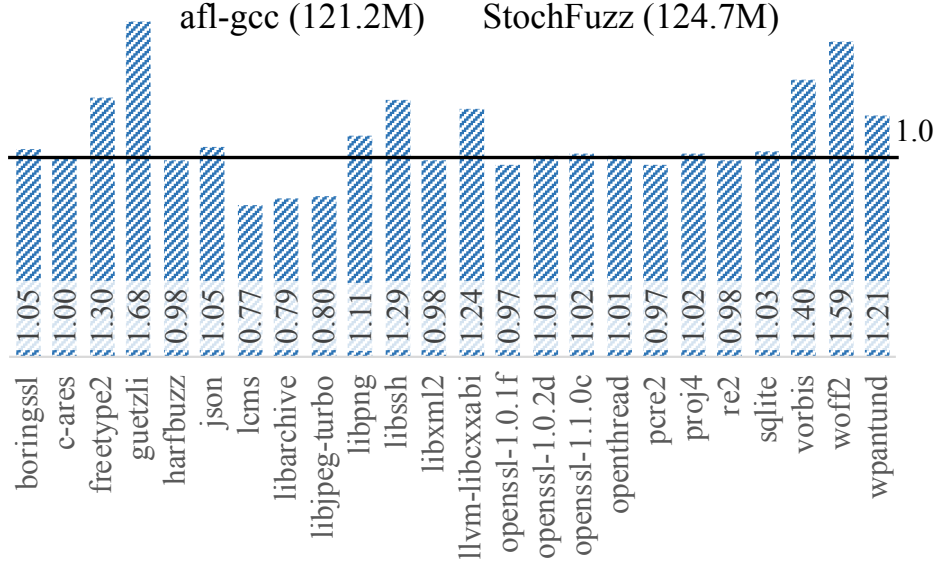


Figure 4.11. Total number of fuzzing executions in 24 hours on Google FTS with intentional data inlining

on the fuzzing results and hence rewriting errors in them are irrelevant to our purpose. And as long as they are covered/accessed, STOCHFuzz can expose and repair their rewriting errors. The results are presented in Table 4.7. The second column presents the number of instrumented basic blocks. Columns 3-6 present the numbers of intentional crashes caused by `hlt` (indicating discovery of new code), unintentional crashes caused by rewriting errors, and unintentional crashes caused by program bugs, and their sum, respectively. The last four columns show the percentage of FN and FP at the beginning and the end of fuzzing process. Observe that at the beginning, with the initial probability analysis results, STOCHFuzz has 11.74% FNs and 1.48% FPs on average. At the end, they are reduced to almost non-existent (0.04% and 0.03%, respectively). These results are consistent with our theoretical bounds developed in Section 4.4. We randomly inspect some of the FPs and find that all of them are data bytes that have no effect on execution path (and hence have no negative impact on fuzzing results). Neither do they cause crashes. Also note that the FNs are at the byte level. If we look at the basic block level, STOCHFuzz does not miss any basic blocks. In other words, in very rare cases (0.04%), it may miss the first one or two bytes in a basic block, but recognizes and instruments the following instructions. These FNs hence have no impact

Table 4.7. Incremental and stochastic rewriting. #IC, #UCE, #UCB, and Sum denote the number of intentional crashes, unintentional crashes caused by rewriting errors, unintentional crashes caused by real bugs, and their sum, respectively. FN and FP denote false negative and false positive, respectively. “Begin” and “End” denote the beginning and end of fuzzing.

Program	Crashes				Rewriting			
	#IC	#UCE	#UCB	Sum	Begin		End	
					%FN	%FP	%FN	%FP
boringssl	114	98	0	212	12.59	6.18	0.09	0.08
c-ares	2	0	0	2	17.49	0.00	0.00	0.00
freetype2	335	10	0	461	10.58	2.47	0.03	0.05
guetzli	200	1	0	201	8.46	0.16	0.01	0.00
harfbuzz	448	5	0	453	9.25	4.64	0.04	0.14
json	80	0	0	80	14.40	0.00	0.02	0.00
lcms	137	0	0	137	16.90	0.04	0.06	0.01
libarchive	215	0	0	215	11.35	0.00	0.04	0.00
libjpeg-turbo	77	4	0	81	9.11	2.91	0.03	0.26
libpng	32	0	0	32	8.17	0.00	0.01	0.00
libssh	123	1	0	124	19.56	0.09	0.04	0.00
libxml2	315	2	0	317	8.80	0.05	0.04	0.00
llvm-libcxxabi	304	0	7,258	7,562	12.86	0.00	0.00	0.00
openssl-1.0.1f	166	45	0	211	12.29	0.50	0.18	0.01
openssl-1.0.2d	25	3	0	28	9.74	0.00	0.03	0.00
openssl-1.1.0c	183	186	0	369	11.11	2.61	0.13	0.08
openthread	19	7	0	26	13.77	0.37	0.06	0.00
pcre2	398	2	37	437	5.64	0.97	0.00	0.00
proj4	46	1	0	47	13.16	0.14	0.02	0.00
re2	133	2	0	135	18.80	0.37	0.09	0.02
sqlite	693	7	0	700	9.03	0.31	0.02	0.00
vorbis	51	7	0	58	8.74	3.25	0.04	0.10
woff2	33	19	0	52	5.31	10.45	0.02	0.04
wpantund	893	1	0	894	14.53	0.00	0.05	0.00
Average	209	17	304	535	11.74	1.48	0.04	0.03

on fuzzing results. Also observe that the number of crashes by rewriting errors is very small (17) compared to that of intentional crashes (209). The former entails the relatively more expensive error diagnosis and repair process. It implies that most rewriting errors are fixed by observing new coverage, without triggering unintentional crashes. Figure 4.12 shows how these numbers change over time for *freetype2*. Observe that they stabilize/converge quickly. The results for others are similar and elided.

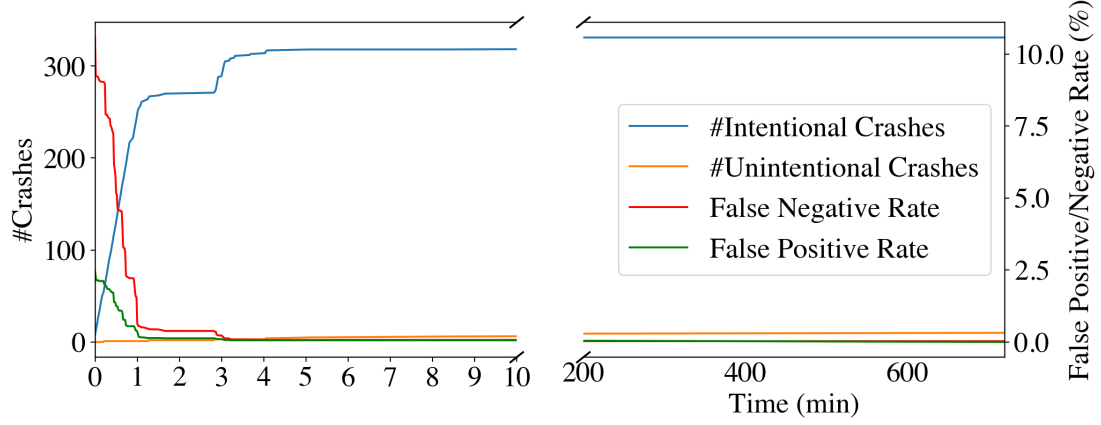


Figure 4.12. Change of intentional/unintentional crashes and false positive/negative rate over time for *freetype2*

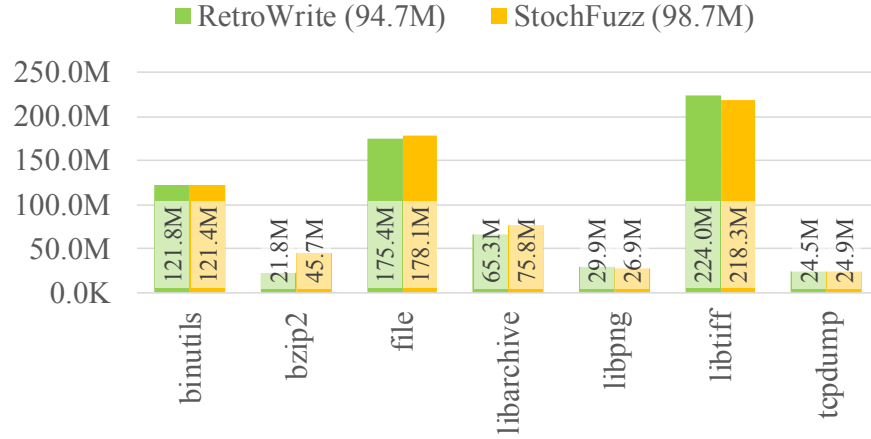


Figure 4.13. The number of total fuzzing executions in 24 hours on *RetroWrite*’s fuzzing benchmarks

4.6.3 Comparison with RetroWrite

Different from other techniques, *RetroWrite* has a number of strong prerequisites about target binaries. The binary has to contain symbols and relocation information, should not be written in C++, should not contain inlined data, and is position independent. Hence, *RetroWrite* cannot be used in the Google FTS experiments. To compare with *RetroWrite*, we use their benchmarks that satisfy all the above conditions. Figure 4.13 and Table 4.8

Table 4.8. Path Coverage on *RetroWrite*’s benchmark

Tools	binutils	bzip2	file	libarchive	libpng	libtiff	tcpdump	Average
<i>RetroWrite</i>	6200	636	29	2706	977	969	3673	2170
STOCHFuzz	6392	1416	29	2384	928	969	3344	2209

Table 4.9. Zero-day vulnerabilities disclosed by STOCHFuzz

Program	Released Date	Size	MD5	Status
CUDA Binary Utilities	2020-09-20	33M	edaf12b5	Fixed
PNGOUT	2020-01-15	89K	64f6899d	CVE-2020-29384

show the numbers of fuzzing executions and the path coverage in 24 hours, respectively. STOCHFuzz led to 98.7M executions and *RetroWrite* led to 94.7M executions, on average. The results show STOCHFuzz achieves similar performance to *RetroWrite*.

4.7 Case Studies

4.7.1 Finding Zero-days in Closed-source Programs

In this case study, we demonstrate STOCHFuzz’s applicability in closed-source or COTS binaries. We run STOCHFuzz on a set of 7 such binaries including CUDA Toolkit (*cuobjdump*, *nvdasm*, *cu++filt*, and *nvprune*), PNGOUT, RAR (*rar* and *unrar*) for a week. It discloses two zero-day vulnerabilities, as listed in Table 4.9. The first column presents the programs, and columns 2-5 present the release date of subject programs, the size, the first 4 bytes of MD5 Hash, and current bug status, respectively. CUDA Binary Utilities, developed by NVIDIA, are a set of utilities which can extract information from CUDA binary files [114]. The bug has been Fixed in CUDA 11.3 [115]. PNGOUT is a closed-source PNG file compressors, which is adopted by multiple commercial or non-commercial image optimizers [116, 117]. These optimizers are further used by thousands of website to speed up image uploading. The PNGOUT vulnerability has been assigned a CVE ID.

Table 4.10. Maze solving by different approaches

Maze		Plain	IJON-Source	IJON-Binary	
		<i>afl-clang-fast</i>	<i>afl-clang-fast</i>	<i>afl-qemu</i>	STOCHFuzz
Small	Easy	2/3	✓	✓	✓
	Hard	1/3	✓	✓	✓
Large	Easy	✗	✓	✓	✓
	Hard	✗	✓	✓	✓

4.7.2 Collect Other Runtime Feedback Than Coverage

We conduct a case study in which we use STOCHFuzz to collect other runtime feedback than coverage. IJON [95], a state-aware fuzzing technique, increases fuzzing effectiveness by observing how the values of given variables change. Specifically, the tester annotates important variables in source code and the compiler instruments accesses to these variables to track their runtime changes. The changes, together with code coverage, guide input mutation. As reported in [95], it substantially improves fuzzer performance for specific kinds of programs such as complex format parsers. We port IJON to support binary-only fuzzing based on STOCHFuzz, and conduct the same maze experiment in the IJON paper, which was used to show the effectiveness of state-aware fuzzing. In the experiment, the target programs are games where the player has to walk through an ASCII art maze. Fuzzers instead of a human player are used to walk the mazes. IJON has advantages over vanilla fuzzers as it observes maze states and uses them to guide input mutation. The ported IJON can resolve the mazes as fast and as effective as the original source-based version, and much more effective than running IJON on *afl-qemu*.

We follow the exact same setup in IJON, with two maze sizes (large and small) and two sets of rules. With the easy rule, a game is terminated once an incorrect step is taken. With the hard one, the player is allowed to backtrack. Note that in the later case, the state space is much larger. We experiment with 4 tools, *afl-clang-fast* without IJON plugin, *afl-clang-fast* with IJON plugin, binary-only *afl-qemu* with ported IJON plugin, and binary-only

Table 4.11. Average time-to-solve in minutes \pm the standard deviation to solve the small / large maze

Maze		Plain	IJON-Source	IJON-Binary	
		<i>afl-clang-fast</i>	<i>afl-clang-fast</i>	<i>afl-qemu</i>	STOCHFuzz
Small	Easy	95.42 \pm 40.47	1.52 \pm 0.45	20.96 \pm 10.56	1.64 \pm 0.51
	Hard	149.78 \pm 0.0	0.46 \pm 0.09	3.85 \pm 1.90	0.52 \pm 0.06
Large	Easy	-	20.66 \pm 9.19	150.28 \pm 30.27	22.94 \pm 14.49
	Hard	-	5.31 \pm 1.59	96.85 \pm 16.61	5.12 \pm 1.89

STOCHFuzz with ported IJON plugin. We run each tool three times with a 12-hour timeout according to the setting of the original paper. Table 4.10 shows the overall effectiveness. The first column presents the different mazes under different rules. Columns 2-5 denote whether the maze is solved by the 4 different tools, respectively. Symbol \times denotes no solution was found in any run, \checkmark denotes that all runs solved the maze. *Afl-clang-fast* solves the small maze with the easy rule 2 out of 3 trials, and the small maze with the hard rule 1 out of 3 trials. The other tools successfully solve all the mazes. Table 4.11 shows the average time (in minutes) needed to solve the mazes and the standard deviation. Observe that although *afl-clang-fast* can solve some small mazes, it takes the longest time. Regarding the two binary-only approaches, STOCHFuzz is around 8 \times faster than *afl-qemu*. Additionally, STOCHFuzz only has around 8% slowdown compared with *afl-clang-fast* plus IJON, which demonstrates the capabilities of STOCHFuzz.

4.8 Summary

We develop a new fuzzing technique for stripped binaries. It features a novel incremental and stochastic rewriting technique that piggy-backs on the fuzzing procedure. It leverages the large number of trial-and-error chances provided by the numerous fuzzing runs to improve rewriting accuracy over time. It has probabilistic guarantees on soundness. The empirical results show that it outperforms state-of-the-art binary-only fuzzers that are either not sound or having higher overhead.

5. EXPANDING VIEWPOINTS: DELVING INTO DL-BASED BINARY ANALYSIS

Deep Learning (DL) models are increasingly used in many cyber-security applications and achieve superior performance compared to traditional solutions. In this chapter, we discuss backdoor vulnerabilities in naturally trained models used in binary analyses. These backdoors are not injected by attackers but rather products of defects in datasets and/or training processes. The attacker can exploit these vulnerabilities by injecting some small fixed input pattern (e.g., an instruction) called backdoor trigger to their input (e.g., a binary code snippet for a malware detection DL model) such that misclassification can be induced (e.g., the malware evades the detection). We focus on transformer models used in binary analysis. Given a model, we leverage a trigger inversion technique particularly designed for these models to derive trigger instructions that can induce misclassification. During attack, we utilize a novel trigger injection technique to insert the trigger instruction(s) to the input binary code snippet. The injection makes sure that the code snippets' original program semantics are preserved and the trigger becomes an integral part of such semantics and hence cannot be easily eliminated.

5.1 Introduction

Rapidly advancing Deep Learning (DL) techniques have led to unprecedented capabilities in many areas, such as Computer Vision (CV), Natural Language Processing (NLP), and Robotics. Many believe that similar new capabilities can be developed for cyber-security applications. Recently, DL models are being increasingly used in a wide range of security tasks, such as *binary code disassembly* for malware analysis and code hardening [101, 118–120], *binary similarity analysis* for malware detection, software fingerprinting, and code theft detection [121–128], *decompilation* including type inference [129–132], function signature inference [129, 131–133], and function name prediction [131, 134–136], APT attack forensics [137–142], and intrusion detection [143–146]. These techniques demonstrate superior performance compared to their traditional counterparts that are not based on DL models. The advan-

tages of using such data-driven techniques are clear. In particular, many cyber security tasks have substantial inherent uncertainty. For example, a classic challenge in decompilation is to determine variable types when symbolic information has been stripped away. To recover such types, many heuristics have to be used, leading to uncertainty. Such uncertainty can be naturally modeled by probabilities [45] and reasoned by distribution analysis, which are the underpinnings of DL techniques. In addition, while rules and heuristics used in classic techniques require substantial domain expertise, DL techniques can automatically learn such rules from data. For example, XDA [118] and DeepDi [119] are recent proposals that use DL models to recognize function entries in binary executables and then perform disassembly. They do not require any pre-defined rules or heuristics. Instead, they train DL models from a large code repository and achieve superior performance. Inspired by these successes, many more DL based security solutions will likely be developed and deployed in the near future.

However, recent research [147, 148] in the CV and NLP domains have demonstrated that pre-trained clean DL models are vulnerable to *backdoor attack* [149–154], which is a special kind of *adversarial attack* [155–160]. Specific input pattern called *backdoor trigger* can be derived such that samples stamped with such trigger can cause the model to misbehave, e.g., misclassify to some *target label*. These triggers are usually model specific but not input specific. Examples of triggers include a small patch (for vision models) and a special word (for NLP models). In contrast, adversarial attacks [155, 157] derive unique perturbations for individual samples to induce misclassification and hence are input specific.

The study of backdoor vulnerabilities in naturally trained deep learning models used in binary analysis tasks is of importance in the development of cutting-edge cyber-security solutions. The exploitation of such vulnerabilities can have severe consequences, particularly in malware analysis. Despite the utilization of deep learning models, human analysts still play a vital role in the analysis of malware samples and tracing their origins. Security companies, such as Mandiant [161], have made substantial investments in the development of reverse engineering tools specifically for use by human analysts, including tools for symbol recovery, function annotation, binary code matching, and binary code attribution. If these models were to be attacked and produce incorrect labels, such as manipulated function names, it could lead to human analysts overlooking critical attack behaviors and ultimately failing in

their analysis tasks. This highlights the importance of ongoing research into potential attack techniques and the improvement of the security of these models. It is also worth noting that there is a substantial body of existing work [149–154, 162–164] on injecting backdoors into deep learning models through poisoned training data. However, in our context, we are more focused on finding backdoor vulnerabilities in models trained naturally, referred to as *natural backdoors*. This is because security models are usually trained by trusted parties. Existing attacks in the vision and NLP domains cannot be easily adapted to attack these models. Specifically, vision models deal with a continuous input space, namely, input pixels can change continuously. Hence, existing attacks often leverage gradient descent to invert a backdoor trigger. In contrast, many security DL models deal with discrete inputs, e.g., instruction sequences and log entries. Continuous input changes unlikely yield new valid inputs. For example, changing the encoding of a `ret` instruction `0xC3` to `0xC4` does not yield a valid instruction. NLP models deal with similar discrete inputs, which need to be a sequence of legitimate words. Existing attack methods in the NLP domain mitigate the problem by inverting triggers in the continuous word embedding domain instead of the discrete input domain and finding the input that has the closest embedding to the inversion result [165, 166]. However, backdoor triggers generated for security models often need to preserve strict semantic properties when inserted to an input. For example, a `mov` trigger instruction may completely break the semantics of a malware when inserted.

In this chapter, we develop a novel method to identify and exploit backdoor vulnerabilities in DL models used in recent binary analysis models. These models take binary executable code as input and predict various things such as instruction boundaries, function entries, function signatures, and code similarities. They serve a wide range of downstream cyber security applications. Our attack is effective and successfully compromises all the models we study, including some closed-source models that run as commercial online services. By exploiting the backdoors identified by our technique, the attacker can mutate their binaries accordingly (using our tool) before releasing them to the wild and the mutated binaries can fail model-based disassembly/decompilation efforts, disrupt analysis, and so on. Our attack features a trigger inversion method that can guarantee the generated triggers are legitimate instruction sequences. It also has a novel trigger insertion method that not only preserves the

semantic of an input binary, but also ensures that *the trigger instruction becomes part of the original semantics after injection*, instead of inaccessible code that can be easily identified and removed.

Our contributions are summarized as follows.

- We study backdoor vulnerabilities in naturally trained DL models used in binary code analysis. Our findings suggest that such vulnerabilities widely exist and they need to be properly mitigated due to their critical roles in security applications.
- We develop a trigger inversion technique that can generate valid instructions as backdoor triggers.
- We devise a trigger injection technique that ensures the trigger becomes an integral part of the original code’s semantics and the injected (and patched) code has the same semantics as before. It features a block-level randomized execution engine and a symbolic patching method.
- We develop a prototype PELICAN [167] and evaluate it on 5 binary analysis tasks and 15 models. Our evaluation shows that PELICAN can achieve 86.09% attack success rate (ASR) with only three trigger instructions. PELICAN has 93.01% higher ASR than a baseline method that adapts an existing NLP trigger inversion technique; 94.14% of injected triggers by PELICAN can evade detection, whereas all the triggers injected by opaque predicates [168] are detected. Our backdoor-injected binaries have 204.23% lower runtime overhead compared to those by opaque predicates. We also conduct a case study of exploiting two closed-source commercial tools, i.e., DeepDi [119] and BinaryAI [128], in the black-box scenario. PELICAN will be publicly available upon publication.

Threat Model. We aim to exploit backdoors in naturally trained models, not models that have injected backdoors by data poisoning [149] or trojanning [169]. This is analogous to finding vulnerabilities in regular software, not malware. We focus on transformer models used in binary code analyses, which are primitives for a wide range of cyber security applications: malware analysis, vulnerability finding, software hardening, decompilation, and foren-

sic analysis. Transformers are the most effective models in these analysis, out-performing other models such as CNN, RNN, and LSTM. Note that attacking models used in other applications, such as network traffic based intrusion detection requires completely different trigger injection technique (in order to preserve traffic semantics). We hence consider it out of the scope.

We consider two scenarios: *white-box attack* and *black-box attack*. In the former, we assume the attacker has access to the model such that gradient descent can be applied to generate trigger instructions. Note that many binary analysis tools (and hence the DL models used by these tools) [15, 39, 119, 170] are supposed to run by the end users. It is hence reasonable to assume the attacker can access these models. Even if these tools are closed-source, the attacker can still leverage model reverse engineering techniques [171–176] to acquire model copies. In the black-box attack scenario, the attacker does not have access to the subject model. We hence leverage the transferability [177, 178] of these backdoor vulnerabilities. The assumption is that many these models tend to learn similar features, which are rooted at the compiler behaviors, e.g., function epilogue and prologue, leading to similar vulnerabilities. As such, the attacker can derive a backdoor trigger on a model he has access to, and then use that to exploit another model that he has no access to. In addition, although not explored in this chapter, black-box attacks that utilize gradient approximation [179] can be leveraged too. We will leave it to our future work. At the end, we want to point out our trigger injection technique is general, applicable in both white-box and black-box scenarios.

Natural Backdoor and Universal Perturbation. Most existing backdoor attacks require data poisoning to inject a trigger. During attack, stamping the trigger can universally cause misclassification for many inputs. We call the problems identified in this chapter *natural backdoor* as we can find a trigger in naturally trained models that can be exploited in the same way as those in injected backdoors. Natural backdoor shares a similar nature as *universal adversarial perturbations* [180] which was originally proposed in the CV domain. Specifically, a universal adversarial perturbation that is small and pervasive can cause misclassification of the subject model. We call the backdoor we study natural backdoor to raise the alert level as it is analogous to vulnerabilities in software.

```

1.  typedef struct entry_t {
2.      struct entry_t *next;
3.      struct entry_t *prev;
4.      int data;
5.  } Entry;
6.
7.  void init_data(Entry *p, int x)
8.  {
9.      p->data = x;
10. }
11. void init_auth_entry(
12.     Entry at[], int i)
13. {
14.     Entry *p = &at[i];
15.     Entry *q = &at[i + 1];
16.     p->next = q;
17.     q->prev = p;
18.     init_data(p, 0);
19. }
20.

```

(a) Source code of the motivation example

```

<init_data>:
A1. push    rbp
A2. mov     rbp, rsp
A3. mov     qword ptr [rbp-8], rdi # store p into a local variable
A4. mov     dword ptr [rbp-12], esi # store x into a local variable
A5. mov     rax, qword ptr [rbp-8] # load p into register rax
A6. mov     edx, dword ptr [rbp-12] # load x into register edx
A7. mov     dword ptr [rax+16], edx # p->data = x
A8. pop     rbp
A9. ret

```

(b) Assembly code of `init_data` compiled w/ O0

```

<init_auth_entry>:
B1. movsxd  rax, esi # store i into a register rax
B2. lea     rax, [rax+rax*2] # rax *= 3 (rax = 3 * i)
B3. shl     rax, 3 # rax <=< 3 (rax = 24 * i)
B4. lea     rdi, [rdi+rax] # rdi = p = &s[i] (sizeof(*p)=24)
B5. lea     rsi, [rdi+24] # rsi = q = &s[i + 1]
B6. mov     qword ptr [rdi], rsi # p->next = q
B7. mov     qword ptr [rsi+8], rdi # q->prev = p
B8. mov     esi, 0
B9. call    init_data # init_data(p, 0)
B10. ret

```

(c) Assembly code of `init_auth_entry` compiled w/ O3

Figure 5.1. Motivation Example

5.2 Motivation

We use an example to motivate our technique. In this example, we use a transformer based technique StateFormer [129] to reverse engineer the function signatures (i.e., function parameters and their types) of code snippets from the leaked *Linux.Mirai* malware [181].

It is an important step for downstream tasks, such as understanding the malware behavior and malware classification. We then show how a naive attack adapted from an existing transformer attack in the NLP domain has difficulties and how PELICAN addresses these challenges.

Figure 5.1 (a) presents a source code snippet from the malware [181], which is simplified for the illustrative purpose. We show the source code just for better understanding and all the tools in this chapter work directly on stripped binaries. Specifically, lines 1-5 declare a doubly linked list. Function `init_data` (lines 7-10) updates the data field of `*p` by a given integer `x`. On the right side, function `init_auth_entry()` initializes the `i`-th element of a given `Entry` array `at` (lines 11-12), linking it with the $(i + 1)$ -th element in `at` and setting its `data` field to 0. Figure 5.1 (b) and Figure 5.1 (c) show the assembly code of `init_data()` and `init_auth_entry()`, respectively. We comment each instruction with its corresponding source code information for interested readers. We compile `init_data()` with O0 and `init_auth_entry()` with O3. We will show that we can exploit an backdoor to cause StateFarmer to produce wrong function signatures for both functions without changing their semantics.

Transformer Pipeline of StateFormer StateFormer is a transformer-based binary type inference technique that can recover precise function signatures from stripped binaries. It is highly resilient to compiler obfuscation. Figure 5.2 depicts the pipeline. A piece of assembly code, e.g., the `init_auth_entry()` function, is fed to StateFormer as input and tokenized as an assembly sequence at step 1. Each assembly instruction is split into multiple tokens during tokenization, e.g., the first instruction `movsxd rax, esi` is tokenized to four tokens “MOVSD”, “RAX”, “,”, and “ESI”. At step 2, to avoid a prohibitively large vocabulary size, StateFormer abstracts away all the immediate values in instructions, e.g., the token “2” in the second instruction `lea rax, [rax+rax*2]` is normalized as “NUM”. Note that the token “`init_data`” in instruction `call init_data` has also been normalized, since function addresses are encoded as immediate values in the machine code. A transformer model then predicts the function signature from the normalized assembly sequence at step 3.

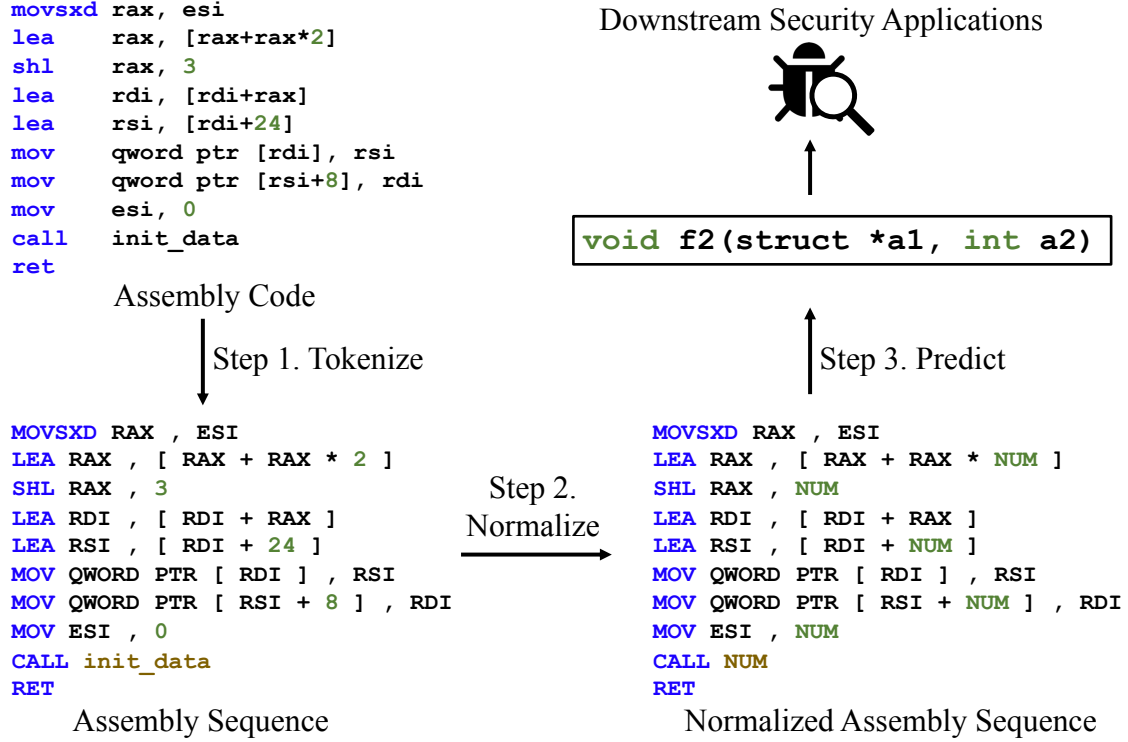


Figure 5.2. Pipeline of StateFormer

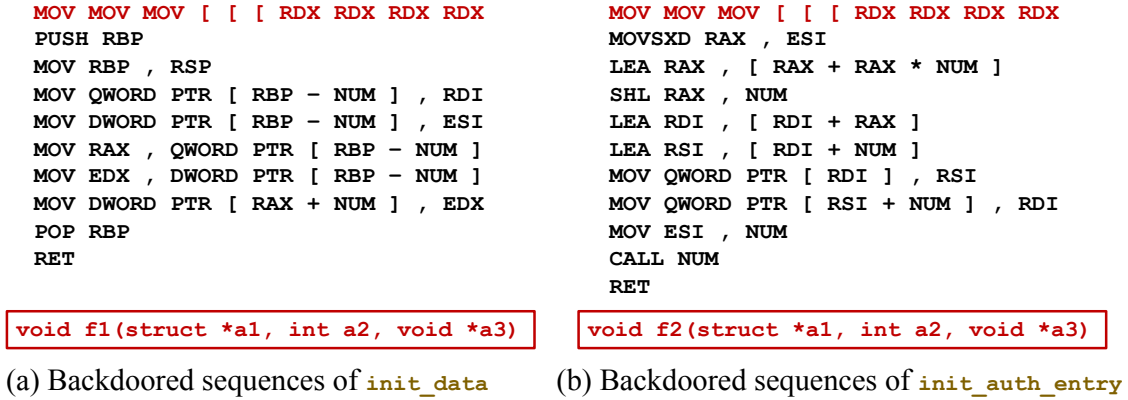


Figure 5.3. Natural backdoor generated by an existing NLP trigger inversion technique

Natural Backdoor in StateFormer by Existing Attack Technique. As demonstrated by a recent study [147, 182], natural backdoors are prevalent in the computer vision and NLP domains, even in naturally trained clean models. It is often due to the model being overfitted on some low-level features [147] We speculate similar vulnerabilities may exist in

models used in binary analysis. Intuitively, mainstream compilers tend to introduce specific code patterns, e.g., field accesses are performed by first loading the base address of data structure to a register, and then adding the field offset to the register. These low-level syntactic code patterns are prevalent in the training set, likely causing overfit.

We first adapt a state-of-the-art adversarial attack for transformer models in the NLP domain [183] to scan for possible natural backdoors in StateFarmer. The technique uses gradient descent to invert some tokens that can cause the model to misbehave on all the given input samples when they are stamped with the tokens. Figure 5.3 presents the inverted trigger, where Figure 5.3 (a) and Figure 5.3 (b) show the backdoored assembly sequences for functions `init_data()` and `init_auth_entry()`, respectively, with the trigger highlighted in red at the beginning and the mis-predicted function signatures at the bottom. Note that although we only present two functions, the inverted trigger can alter the results for over 90% of functions. Compared with the ground truth, the mis-predicated function signature contains one more argument `void *a3`.

With further inspection, we find that register `rdx` is used to pass the third argument (i.e. `a3`) in the x64 calling conventions [184]. We further observe that an assembly instructions with a “MOV” token and a bracket token always performs a dereference of memory address, e.g., `mov [rax], rbx`. Hence, it is likely that StateFormer mistakenly associates these low-level syntactic features with a pointer being passed as `a3`, without considering real program semantics.

However, even though tokens can be found to cause model misbehaviors, the inverted triggers are not legitimate instructions and hence unable to launch a real-world attack, not to mention preserving input code semantics.

Our technique. PELICAN is inspired by two insights.

First Insight: different from natural languages and high-level programming languages (e.g., Java), assembly code has relatively simple syntax. Taking the syntax into account during backdoor generation can produce legitimate trigger instructions without scarifying the efficacy. The syntax of assembly code imposes constraints largely within a single instruction. Our empirical study shows that an individual x64 instruction only comprises 9 tokens on average

Backdoor Trigger: `MOV QWORD PTR [RSI - NUM] , <REG>`

<pre> MOV QWORD PTR [RSI - NUM] , RDI PUSH RBP MOV RBP , RSP MOV QWORD PTR [RBP - NUM] , RDI MOV DWORD PTR [RBP - NUM] , ESI MOV RAX , QWORD PTR [RBP - NUM] MOV EDX , DWORD PTR [RBP - NUM] MOV DWORD PTR [RAX + NUM] , EDX POP RBP RET </pre>	<pre> MOV QWORD PTR [RSI - NUM] , RSI MOVSXD RAX , ESI LEA RAX , [RAX + RAX * NUM] SHL RAX , NUM LEA RDI , [RDI + RAX] LEA RSI , [RDI + NUM] MOV QWORD PTR [RDI] , RSI MOV QWORD PTR [RSI + NUM] , RDI MOV ESI , NUM CALL HEXVAR RET </pre>
---	---

void f1(struct *a1, void *a2)

void f2(struct *a1, void *a2)

(a) Effectiveness of semantic-breaking trigger injection

<pre> push rbp mov rbp, rsp mov qword ptr [rbp-8], rdi mov dword ptr [rbp-12], esi mov rax, qword ptr [rbp-8] mov edx, dword ptr [rbp-12] mov dword ptr [rax+16], edx pop rbp ret </pre>	<pre> xchg rsi, rsp mov qword ptr [rsi-8], rdi mov dword ptr [rsi-12], esp mov rax, qword ptr [rsi-8] mov edx, dword ptr [rsi-12] mov dword ptr [rax+16], edx xchg rsi, rsp ret </pre>
---	--

void f1(struct *a1, int a2)

void f1(struct *a1, void *a2)

(b) Effectiveness of semantic-preserving trigger injection for `init_data`

<pre> movsxd rax, esi lea rax, [rax+rax*2] shl rax, 3 lea rdi, [rdi+rax] lea rsi, [rdi+24] mov qword ptr [rdi], rsi mov qword ptr [rsi+8], rdi mov esi, 0 call init_data ret </pre>	<pre> movsxd rax, esi lea rax, [rax+rax*2] shl rax, 3 lea rdi, [rdi+rax] lea rsi, [rdi+24] mov qword ptr [rsi-24], rsi mov qword ptr [rsi+8], rdi mov esi, 0 call init_data ret </pre>
---	--

void f1(struct *a1, int a2)

void f1(struct *a1, void *a2)

(c) Effectiveness of semantic-preserving trigger injection for `init_auth_entry`

Figure 5.4. Natural backdoor generation and semantics-preserving trigger injection by PELICAN

and 17 tokens at most. As such, we devise a syntax-aware trigger inversion technique taking advantage of a pre-defined instruction dictionary. The instruction dictionary is collected

from the SPEC2000 dataset (compiled with a large number of different options) and contains 119640 normalized instructions. The trigger optimization is performed over the instruction dictionary instead of individual tokens, and hence intrinsically follows the syntax of assembly code. Figure 5.4(a) presents the instruction sequence with our inverted trigger. The trigger and the mis-predicted function signatures are highlighted at the top and the bottom of the figure, respectively. Observe the second argument is misclassified. In real-world scenarios, three or more trigger instructions may be needed to launch an effective attack, preventing a naive approach of enumerating all instructions in the dictionary, as the complexity of enumerating three instructions is $119640^3 \approx 1.7 \times 10^{15}$.

Second Insight: the injected triggers should not only preserve the program semantics but also become an integral part of the semantics. With triggers in the form of valid instructions, a naive approach to injecting these triggers is to add them as dead-code (and hence having the original input semantics preserved), e.g., using opaque predicates [168]. However, such dead code can be easily detected and eliminated [185–191].

We hence propose a novel injection technique. It is driven by a randomized micro-execution technique that describes program semantics by a set of constraints, and a solving-based synthesis technique that generates code satisfying both the requirements of preserving semantics and injecting triggers (see Section 5.5). Figure 5.4(b) and Figure 5.4(c) illustrate the generated code of `init_data()` and `init_auth_entry()`, respectively, after the semantic-preserving trigger injection. We use red to denote the triggers and blue to denote the other entailed patches. In Figure 5.4(b), to inject the trigger `mov qword ptr [rsi-8], rdi`, the first instruction in blue exchanges the values of registers `rsi` and `rsp`. The next four instructions retain their original functionalities of manipulating the local variables using `rsi` (in blue) as the stack frame pointer. The seventh instruction remains unchanged and the eighth one (in blue) exchanges registers `rsi` and `rsp` back before returning to the caller. Observe that the trigger becomes a natural and integral part of the program data flow and hard to remove. In Figure 5.4(c), `mov qword ptr [rsi-24], rsi` is the trigger instance and also the only change compared with the original code, in which the memory operand changes from `[rdi]` to `[rsi-24]`. The modification is guaranteed to be correct since the

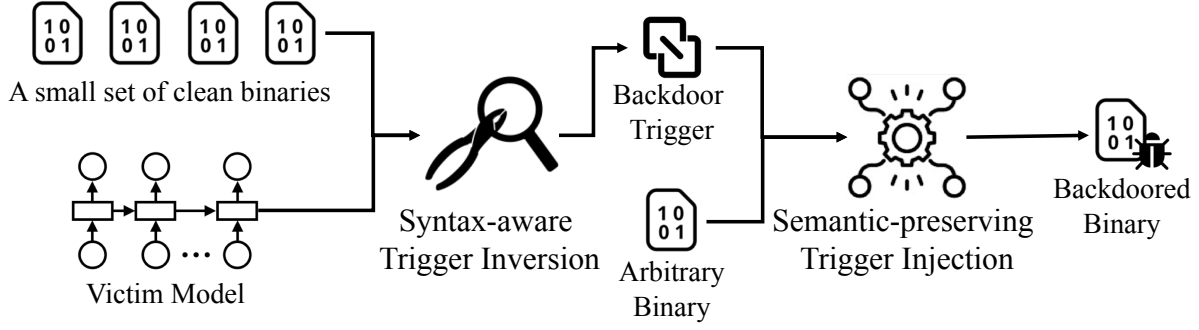


Figure 5.5. Framework of PELICAN

constraint of `rsi = rdi + 24` has been derived from the previous instruction `lea rsi, [rdi+24]`. This illustrates the sophistication of our injection method.

5.3 Design Overview

The overall design is illustrated in Figure 5.5. Given a small set of clean binaries and a victim model, PELICAN reverse-engineers the backdoor trigger using gradient descent. The trigger inversion procedure is syntax-aware. That is, each generated trigger instruction follows the proper assembly syntax. PELICAN achieves this goal by constructing an instruction dictionary, where instructions serve as the backdoor trigger candidates. It then leverages gradient descent to search for the trigger instructions that can induce misclassification. The search by its nature is a discrete optimization problem and cannot be directly solved through gradient descent. PELICAN performs a linear relaxation and defines a convex hull for feasible optimization. See detailed discussion in Section 5.4. With the generated backdoor trigger, the next step is to inject it into some binary without altering the binary’s original semantics. Particularly, given an arbitrary binary, PELICAN first introduces a randomized micro-execution process to extract higher order semantics of the given binary, which are represented by program state changes (e.g., register value changes). These changes are encoded by symbolic constraints. They are resolved together with constraints representing the injection of the trigger and the synthesis of needed patches (to preserve semantics), using Z3 [192] (see details in Section 5.5). Finally, PELICAN produces a backdoored binary with

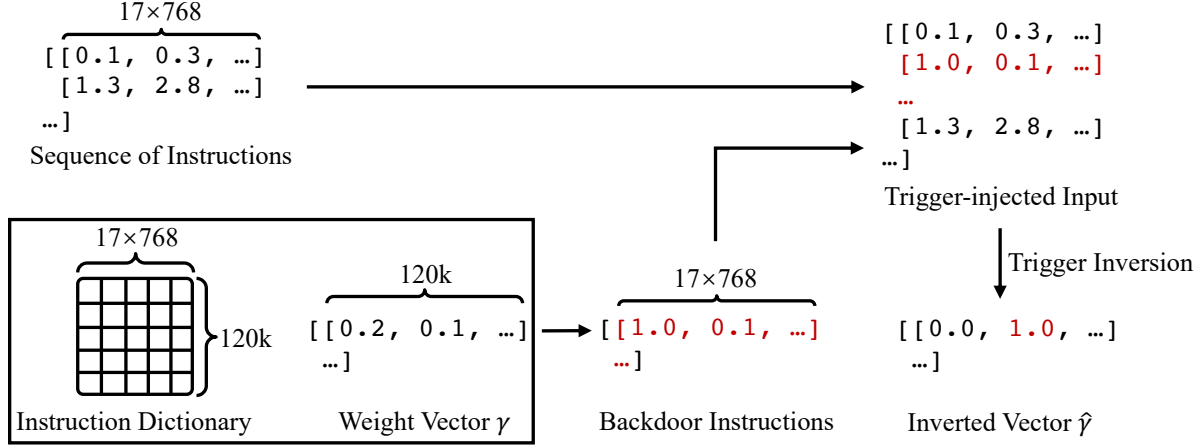


Figure 5.6. Syntax-aware trigger inversion

the trigger injected and the same semantics as the original binary. It can induce the desired misclassification on the victim model.

Use Cases of Pelican. PELICAN operates on assembly code. It leverages datalog disassembly [64], a state-of-the-art binary reassembling tool which has demonstrated its success on thousands of commonly used binaries, to produce reassembleable assembly code from binary. After trigger injection, PELICAN reassembles the code to binary. When source code is available for attackers (e.g., malware developers), PELICAN can also be applied as part of the compilation tool chain, by modifying the intermediate assembly code.

5.4 Syntax-aware Trigger Inversion

Figure 5.6 illustrates the workflow of our syntax-aware trigger inversion. Given an input binary, it is first mapped to the embedding space with 17×768 dimensions for each instruction, where value 17 denotes the number of tokens in an instruction and value 768 the embedding size of a token. The resultant matrix is shown on the top left in Figure 5.6. PELICAN aims to invert a backdoor trigger at the instruction level, which is syntax-aware. We construct an instruction dictionary with 119640 instructions and each instruction is represented by a 17×768 -dimension embedding as shown on the bottom left. PELICAN uses a weight vector γ to denote the trigger, whose size is $u \times 119640$ (where u is the number of

instructions in the trigger and 119640 is the dictionary size). By multiplying the dictionary with γ , PELICAN obtains the trigger instruction embeddings in the middle of Figure 5.6, which is then injected to an input sample on the top right. PELICAN leverages gradient descent to optimize the weight vector γ such that it can induce misclassification for a set of samples. Ideally, the inverted vector $\hat{\gamma}$ has only one dimension with 1 that denotes the trigger instruction and the others with 0. For example, the second dimension having value 1 means that the inverted trigger is the second instruction in the dictionary.

5.4.1 Trigger Generation

The trigger generation aims to produce a small piece of binary code that can induce misclassification on the subject binary analysis model. A straightforward idea is to directly generate code tokens, such as operators, registers, etc. Such a method however cannot guarantee the generated trigger code snippet following the proper syntax of assembly code. As discussed in Section 5.2 and shown in Figure 5.3, the backdoor trigger generated by an existing NLP inversion technique has invalid syntax.

In PELICAN, we construct an instruction dictionary collected from the SPEC2000 dataset containing 119640 normalized instructions. This provides us with a large pool of feasible trigger candidates. We hence make use of a gradient descent method to search for the possible combination of instructions as the backdoor trigger, which can induce misclassification for a set of input binary samples on the subject model. Assume a subject model $f : \mathcal{X} \mapsto \mathcal{Y}$, the instruction dictionary d (i.e., a large table of embeddings), and a discrete variable $c \in N_+^{u \times v}$, where u is the number of instructions in the trigger and v is the size of the dictionary d (i.e., the number of instructions). The trigger generation process can be written as follows.

$$\arg \max_c \mathbb{E}_{(x,y) \sim \{\mathcal{X}, \mathcal{Y}\}} \mathcal{L}\left(f\left(x \oplus d(c)\right), y\right), \quad (5.1)$$

where (x, y) is a sample from the set $\{\mathcal{X}, \mathcal{Y}\}$ that we use for trigger generation. We assume x has already been mapped to the embedding space for discussion simplicity. \mathcal{L} is the loss function. Operator \oplus denotes the trigger stamping. Operation $d(c)$ looks up the instruction embeddings for index c in the dictionary. Observe that index c is discrete and

hence cannot be directly optimized through gradient descent [193–195]. To address the above non-differentiability problem, we construct a convex hull to denote the input space.

Definition 5.4.1. *Let $\mathcal{S} = [1, v]$ be the set of instructions in the dictionary. The convex hull over the input space is $\mathcal{H} = \{\sum_{i=1}^v \gamma_i d(i) \mid \sum_{i=1}^v \gamma_i = 1, \gamma_i \geq 0\}$.*

An input t in the hull is essentially a weighted sum of all instruction embeddings in the dictionary: $t = \sum_{i=1}^v \gamma_i d(i)$ and the sum of weights $\gamma_i, i \in \{1, 2, \dots, v\}$ must equal to 1. To satisfy the constraint, we introduce a weight vector p and compute γ as the *softmax* over p like the following.

$$\gamma_i = \frac{\exp(p_i)}{\sum_{j=1}^v \exp(p_j)}. \quad (5.2)$$

Note that with the projection, while p is unbounded (and hence easy to optimize), γ can satisfy the constraint of summing up to 1. With the above formalization, we avoid optimizing in the discrete index space (variable c in Equation 5.1) but rather focus on the weight vector p and hence γ in the convex hull (which is differentiable). The trigger generation is thus to solve the following optimization problem¹.

$$\arg \max_{\gamma} \mathbb{E}_{(x,y) \sim \{\mathcal{X}, \mathcal{Y}\}} \mathcal{L}\left(f\left(x \oplus \left\{\sum_{j=1}^v \gamma_{ij} d(j)\right\}_{i=1}^u\right), y\right). \quad (5.3)$$

In a nutshell, we use a linear combination of all the possible instructions in the dictionary as the potential trigger instruction and allow the optimization to find the most promising instruction through gradient descent. The ideal ultimate γ has only one dimension with value 1 and the others value 0. In practice, the weight vector may not always converge to an ideal vector. We hence sort the weight values in p in descending order and select the top 5 instructions as the trigger candidates.

Synonym Instructions. We expand the pool of potential trigger candidates by including synonymous instructions of the backdoor trigger as well. In particular, synonymous instructions are derived from the original trigger instruction by substituting a terminal symbol with

¹↑We use γ as the variable to optimize in the equation for notation simplicity, whereas our true variable is p .

```

movsxd rax, esi
lea    rax, [rdx+rax*2]
lea    rax, [rax+rax*2]
shl    rax, 3
lea    rdi, [rdi+rax]
lea    rsi, [rdi+24]
mov    qword ptr [rdi], rsi
mov    qword ptr [rsi+8], rdi
mov    esi, 0
call   init_data
ret

```

```
void f2(struct *a1, int a2, int a3)
```

(a) Before trigger injection

```

movsxd rax, esi
mov    rdx, rax
lea    rax, [rdx+rax*2]
shl    rax, 3
lea    rdi, [rdi+rax]
lea    rsi, [rdi+24]
mov    qword ptr [rdi], rsi
mov    qword ptr [rsi+8], rdi
mov    esi, 0
call   init_data
ret

```

```
void f2(void *a1, int a2)
```

(b) After trigger injection

Figure 5.7. Example trigger by per-instance adversarial attack

a different one. The substituted symbols must be of the same token type, e.g., register tokens “RDI” and “RSI”, operand size tokens “QWORD” and “DWORD”.

Location of Trigger Stamping. To invert position-independent trigger instructions, PELICAN stamps triggers at different locations upon different samples. Those locations are randomly selected prior to the optimization step of trigger inversion, and hence vary among samples. During injection, the place of inserting the trigger instruction(s) is determined by the constraint solver (Section 5.5).

5.4.2 Why Not Per-instance Adversarial Attack

A backdoor trigger targets a set of input samples for inducing misclassification. A similar path would be to generate a per-instance perturbation as in adversarial attacks. Technically this is feasible as the goal is to fool the subject model by adding perturbation (e.g., an extra instruction) to the input. In this chapter, we aim to make the trigger an integral part of the original binary’s semantics such that it cannot be easily discarded by sanitization. The trigger injection process hence requires preserving the semantics of the binary through some transformation (discussed in Section 5.5). The triggers generated by per-instance adversarial attack may not be effective after the injection procedure. Figure 5.7 shows an example trigger generated by per-instance adversarial attack. We use the same code snippet as in Section 5.2 for the trigger generation. Figure 5.7(a) presents the assembly code with the directly

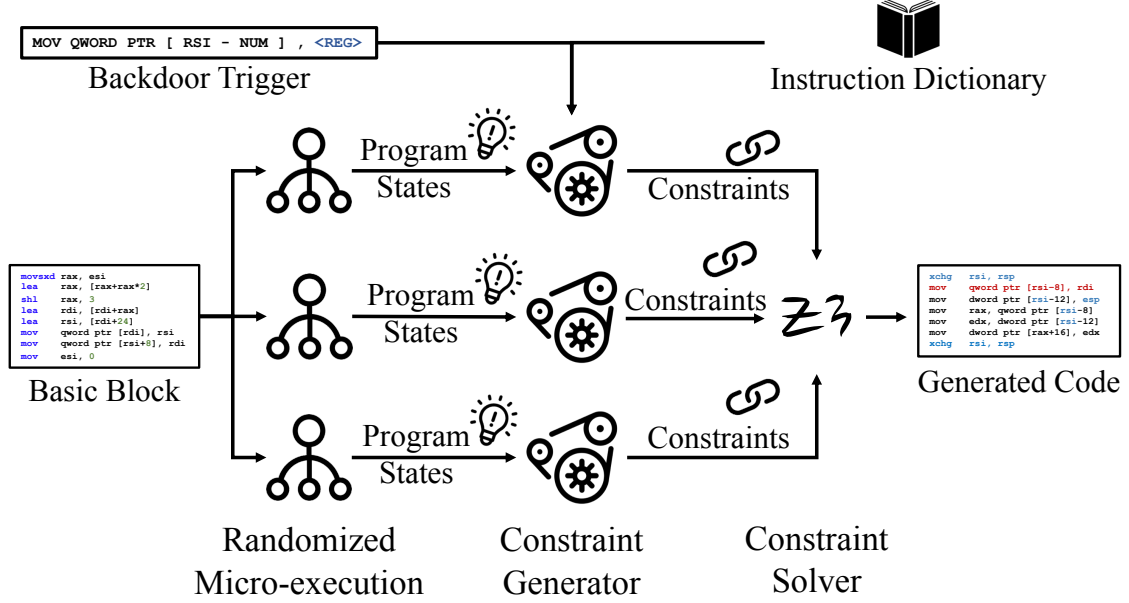


Figure 5.8. Workflow of PELICAN’s semantic-preserving trigger injection

inserted trigger instruction and Figure 5.7(b) the assembly code after semantic-preserving trigger injection. Observe that directly inserting the trigger instruction can cause the subject model to have the wrong prediction. But the trigger is not effective any more after the semantic-preserving transformation. This is because per-instance adversarial attack only aims to induce misclassification for one particular input and the generated trigger may not be effective on other inputs. The assembly code produced by the semantic-preserving transformation is different from the original code, causing the generated trigger by per-instance adversarial attack ineffective. Our generated backdoor trigger, on the other hand, are effective for a set of binary samples, which can maintain its effectiveness on the transformed assembly code.

5.5 Semantic-preserving Trigger Injection

With the triggers in legitimate forms, a naïve approach is to inject the triggers as dead code and hence have the semantics preserved. However, such efforts become ineffective when the input data is sanitized before use, e.g., discarding garbage code or pruning non-critical

program paths. Furthermore, a popular technique to inject dead code, i.e., using opaque predicates [168], usually incurs high runtime overhead, rendering the attack infeasible for some performance-sensitive applications, e.g., crypto-mining malware. We hence propose a novel trigger injection technique. It takes a subject basic block, a backdoor trigger, and a pre-collected instruction dictionary as inputs, and synthesizes a semantic-equivalent code snippet that naturally includes the trigger. In PELICAN, we use a greedy search algorithm to find a basic block to inject trigger. Since these models are usually quite vulnerable, the search is simple and its discussion is elided.

Figure 5.8 depicts the workflow. Given a block, we start multiple micro-executions to extract higher order semantics of the code. The semantics are represented by program states (i.e., the concrete value of each register and memory object) before and after executing the basic block. The constraint generator further transforms the program states to symbolic constraints so that the trigger injection task is reduced to a constraint solving problem. Z3 [192] is hence involved to solve the constraints and synthesize the injected (and patched) code. Note that the code synthesis with a single micro-execution instance is very likely problematic. For example, `mov rax, 12` can be synthesized while the ground truth instruction is `add rax, rax` from a micro-execution result with `rax` being 6 before execution and 12 after. The problem can be avoided with high probability when other micro-execution instances are involved, e.g., having `rax` changed from 37 to 74.

Since the synthesis has only probabilistic guarantees, PELICAN further performs symbolic equivalence checking to ensure the synthesized code is equivalent to the original block. The injection is performed on a different block if the validation is not successful (which is very rare).

An alternative to expressing program semantics is to directly perform symbolic execution (instead of micro-executions then constructing symbolic constraints on *concrete states* like in PELICAN). However, this approach inevitably introduces quantified formulas and array models (for memory objects) in the generated constraints, inducing difficulties in the downstream solver[196–200]. For example, the symbolic constraint of a heap memory read entails an quantifier operation on the symbolic address, which is further translated inside the solver to numerous comparisons with all the possible addresses that have been written to. It is

$\langle \text{BasicBlock} \rangle$	b	$::= s$
$\langle \text{Statement} \rangle$	s	$::= s_1; s_2 \mid l : i$
$\langle \text{Instruction} \rangle$	i	$::= r_d := e \mid r_d := \mathbf{R}(r_a) \mid \mathbf{W}(r_a, r_v)$
$\langle \text{Expression} \rangle$	e	$::= r \mid c \mid e_1 \text{ op } e_2$
$\langle \text{Label} \rangle$	l	$::= \mathbf{L}_0 \mid \mathbf{L}_1 \mid \mathbf{L}_2 \mid \dots$
$\langle \text{Register} \rangle$	r	$::= \mathbf{r}_0 \mid \mathbf{r}_1 \mid \mathbf{r}_2 \mid \dots$
$\langle \text{Operator} \rangle$	op	$::= + \mid - \mid * \mid \div \mid \text{BitOp} \mid \dots$
$\langle \text{Constant} \rangle$	c	$::= \mathbb{Z}$

Figure 5.9. A simple language for branching-free assembly code sequence

hence very expensive. PELICAN, on the other hand, leverages a randomized concrete representation such that the derived constraints are quantifier-free and can be effectively solved using a quantifier-free bit-vector (QF_BV) theory [192].

In the remainder of this section, we discuss details of individual component and how we address practical challenges.

5.5.1 Randomized Micro-execution

The goal of randomized micro-execution is to have a low cost method to concretize program semantics, i.e., how the values of randomly initialized register and memory objects change after executing the code. To do so, the micro-execution needs to calibrate objects that are accessed during execution, and tracks the changes of these objects.

Language. To facilitate discussion, we introduce a low-level language to model basic blocks in assembly code. The language is designed to illustrate our key idea, and hence omits many irrelevant details. The syntax of is in Figure 5.9. A basic block is constituted by a statement which is either a concatenation of two statements or an instruction. We label the location before an instruction as l (like a program counter). Instruction $r_d := e$ denotes the computation and data movement among registers and e denotes an expression. $\mathbf{R}(r_a)$ and $\mathbf{W}(r_a, r_v)$ model memory read and write operations, respectively, where r_a holds the memory address and r_v holds the value to write. Note that the language does not model branching instructions, which will appear unchanged after injection and remain to be the last instruction of a block.

Definitions. We briefly discuss the definitions used by the semantics of the randomized micro-execution. The formal definitions can be found at the top of Figure 5.10. Specifically, we use Λ_{reg} and Λ_{mem} to denote the register store and the memory store, respectively. Different program points, distinguished by labels, have different register and memory stores. For instance, the initial value of **rax** is denoted as $\Lambda_{reg}(\mathbf{L}_0)(\mathbf{rax})$ and the memory object [52] at the program location \mathbf{L}_3 is denoted as $\Lambda_{mem}(\mathbf{L}_3)(52)$. Register and memory stores constitute the program state Λ .

Semantics. The overarching process is to concretely execute the code and check every object before use. If the object is not initialized, we assign it a random value and record the initial value as part of the initial program state (i.e. $\Lambda_{reg/mem}(\mathbf{L}_0)$). As such, the concertized semantics, i.e., which objects are used and what their initial and final states are, can be determined by accessing the program states before and after the execution (i.e., $\Lambda_{reg/mem}(\mathbf{L}_0)$ and $\Lambda_{reg/mem}(\mathbf{L}_{n+1})$ where \mathbf{L}_{n+1} denotes the last program point).

Figure 5.10 presents the semantics of the randomized micro-execution. Specifically, a concatenated statement is evaluated by evaluating its components in sequential order (rule (1)). Note that “ $\Lambda \models s \Rightarrow \Lambda'$ ” denotes that the program state changes from Λ to Λ' after executing the statement s . Rule (2) defines how the micro-execution evaluates expressions. Given a previous program state Λ and an instruction $r_d := r_1 \text{ op } r_2$, PELICAN first initializes registers r_1 and r_2 by *InitReg* (rule (2), line 2), calculates the outcome v of the expression (line 3), and sets register r_d as v at the next program point (line 4). *InitReg*(Λ_{reg}, l, r) initializes register r at the program point l , whose semantics are defined by rules (5) and (6). If register r has already been initialized at the program point l , Λ_{reg} remains the same (rule (5)). Otherwise, a random value v is used to update register r at l (rule (6), lines 1-2), while the initial program state of r is set as the same value v (line 3). The semantics of memory read and write operations are defined by rules (3) and (4), respectively. Specifically, rule (3) first initializes register r_a and reads its value v_a (line 2), initializes and accesses the memory object located at address v_a (line 3), and updates register r_d (line 4). Memory write operation (rule (4)) is similar to the read operation, and *InitMem* (rules (7) and (8)) is similar to *InitReg*.

Definitions:

$$\begin{aligned}\Lambda_{reg} \in RegisterStore &::= Label \rightarrow Register \rightarrow \mathbb{Z} \\ \Lambda_{mem} \in MemoryStore &::= Label \rightarrow \mathbb{Z} \rightarrow \mathbb{Z} \\ \Lambda \in ProgramState &::= \langle RegisterStore \times MemoryStore \rangle\end{aligned}$$

Semantic Rules:

Given a basic block $b ::= s$, “ $\perp \models s \Rightarrow \Lambda$ ” denotes that s is evaluated to Λ from an empty state.

$$\frac{\Lambda \models s_1 \Rightarrow \Lambda', \Lambda' \models s_2 \Rightarrow \Lambda''}{\Lambda \models s_1; s_2 \Rightarrow \Lambda''} \quad (5.4) \quad \frac{\begin{aligned} &(\Lambda_{reg}, \Lambda_{mem}) = \Lambda \\ &\Lambda'_{reg} = InitReg(InitReg(\Lambda_{reg}, l, r_1), l, r_2) \\ &v = \Lambda'_{reg}(l)(r_1) \text{ op } \Lambda'_{reg}(l)(r_2) \\ &\Lambda' = (\Lambda'_{reg}[next(l) \mapsto \Lambda'_{reg}(l)[r_d \mapsto v]], \Lambda_{mem}) \end{aligned}}{\Lambda \models l : r_d := r_1 \text{ op } r_2 \Rightarrow \Lambda'} \quad (5.5)$$

$$\frac{\begin{aligned} &(\Lambda_{reg}, \Lambda_{mem}) = \Lambda \\ &\Lambda'_{reg} = InitReg(\Lambda_{reg}, l, r_a), v_a = \Lambda'_{reg}(l)(r_a) \\ &\Lambda'_{mem} = InitMem(\Lambda_{mem}, l, v_a), v_v = \Lambda'_{mem}(l)(v_a) \\ &\Lambda' = (\Lambda'_{reg}[next(l) \mapsto \Lambda'_{reg}(l)[r_d \mapsto v_v]], \Lambda'_{mem}) \end{aligned}}{\Lambda \models l : r_d := R(r_a) \Rightarrow \Lambda'} \quad (5.6)$$

$$\frac{\begin{aligned} &(\Lambda_{reg}, \Lambda_{mem}) = \Lambda \\ &\Lambda'_{reg} = InitReg(InitReg(\Lambda_{reg}, l, r_a), l, r_v) \\ &v_a = \Lambda'_{reg}(l)(r_a), v_v = \Lambda'_{reg}(l)(r_v) \\ &\Lambda'_{mem} = InitMem(\Lambda_{mem}, l, v_a) \\ &\Lambda' = (\Lambda'_{reg}, \Lambda'_{mem}[next(l) \mapsto \Lambda'_{mem}(l)[v_a \mapsto v_v]]) \end{aligned}}{\Lambda \models l : W(r_a, r_v) \Rightarrow \Lambda'} \quad (5.7)$$

$$\frac{r \in \underline{dom}(\Lambda_{reg}(l))}{\Lambda'_{reg} = \Lambda_{reg}} \quad (5.8) \quad \frac{\begin{aligned} &r \notin \underline{dom}(\Lambda_{reg}(l)), v = \underline{random}() \\ &\Lambda'_{reg} = \Lambda_{reg}[l \mapsto \Lambda_{reg}(l)[r \mapsto v]] \\ &\Lambda''_{reg} = \Lambda'_{reg}[\mathbf{L}_0 \mapsto \Lambda'_{reg}(\mathbf{L}_0)[r \mapsto v]] \end{aligned}}{\Lambda''_{reg} = InitReg(\Lambda_{reg}, l, r)} \quad (5.9)$$

$$\frac{v_a \in \underline{dom}(\Lambda_{mem}(l))}{\Lambda'_{mem} = \Lambda_{mem}} \quad (5.10) \quad \frac{\begin{aligned} &v_a \notin \underline{dom}(\Lambda_{mem}(l)), v_v = \underline{random}() \\ &\Lambda'_{mem} = \Lambda_{mem}[l \mapsto \Lambda_{mem}(l)[v_a \mapsto v_v]] \\ &\Lambda''_{mem} = \Lambda'_{mem}[\mathbf{L}_0 \mapsto \Lambda_{mem}(\mathbf{L}_0)[v_a \mapsto v_v]] \end{aligned}}{\Lambda''_{mem} = InitMem(\Lambda_{mem}, l, v_a)} \quad (5.11)$$

Figure 5.10. Semantics of the randomized micro-execution

Table 5.1. Running example of the randomized micro-execution

<i>Label</i>	<i>Instr</i>	Λ_{reg}	Λ_{mem}	<u>R</u>
$l = \mathbf{L}_0$	$r_1 := r_0$	$\Lambda_{reg}(\mathbf{L}_0)[r_0 \mapsto 6]$	-	6
		$\Lambda_{reg}(\underline{next}(l)[r_0 \mapsto 6])$	-	
		$\Lambda_{reg}(\underline{next}(l)[r_1 \mapsto 6])$	-	
$l = \mathbf{L}_1$	$r_0 := r_0 + r_1$	$\Lambda_{reg}(\underline{next}(l)[r_0 \mapsto 12])$	-	-
$l = \mathbf{L}_2$	$r_1 := \mathbf{R}(r_0)$	-	$\Lambda_{mem}(\mathbf{L}_0)[12 \mapsto 19]$	19
		$\Lambda_{reg}(\underline{next}(l)[r_1 \mapsto 19])$	$\Lambda_{mem}(\underline{next}(l)[12 \mapsto 19])$	
$l = \mathbf{L}_3$	$\mathbf{W}(r_1, r_0)$	-	$\Lambda_{mem}(\mathbf{L}_0)[19 \mapsto 96]$	96
		-	$\Lambda_{mem}(\underline{next}(l)[19 \mapsto 12])$	
<i>Evaluation result after</i> $(\perp, \perp) \models s \Rightarrow (\Lambda_{reg}, \Lambda_{mem})$				
$\Lambda_{reg} \equiv \{\mathbf{L}_0 \mapsto \{r_0 \mapsto 6\}, \mathbf{L}_4 \mapsto \{r_0 \mapsto 6, r_1 \mapsto 19\}\}$ $\Lambda_{mem} \equiv \{\mathbf{L}_0 \mapsto \{12 \mapsto 19, 19 \mapsto 96\}, \mathbf{L}_4 \mapsto \{12 \mapsto 19, 19 \mapsto 12\}\}$				
<u>R</u> denotes <u>random</u> () invoked at each step.				

Example. Table 5.1 presents an example of randomized micro-execution. The first column lists the labels and the second column the correspond instructions. The next two columns demonstrate the changes of register store Λ_{reg} and memory store Λ_{mem} , respectively. The last column shows the generated random values at each program location. The register and memory stores after the micro-execution are presented at the last row. Specifically, instruction $\mathbf{L}_0 : r_1 := r_0$ moves the value of r_0 to r_1 . Since register r_0 is not initialized, a random value 6 is selected, and registers r_0 and r_1 at the program point \mathbf{L}_1 are updated accordingly. $\Lambda_{reg}(\mathbf{L}_0)(r_0)$ is further synced as 6, denoting that the initial value of r_0 is 6. The next instruction sets r_0 as 12. Instruction $\mathbf{L}_2 : r_1 := \mathbf{R}(r_0)$ performs data movement from the memory object $[r_0]$ to r_1 . Note that r_0 is evaluated as 12 at \mathbf{L}_2 but $[r_0] \equiv [12]$ has not been initialized. Memory location $[12]$ is hence randomly initialized as 19, recorded by $\Lambda_{mem}(\mathbf{L}_0)(12)$. The last memory write operation accesses a new memory object $[r_1] \equiv [19]$ which is initialized as 96 and then updated as 12. Observe that two register objects r_0 and r_1 and two memory objects $[12]$ and $[19]$ are determined to be accessed during execution, based on the final program state. The value changes of these objects are also well demonstrated.

□

5.5.2 Constraint Generation

With the concretized program semantics (i.e., Λ), PELICAN aims to generate a new code snippet that performs the same state updates on register and memory objects, and injects the trigger instructions. To do so, it first introduces a set of boolean variables x_i^k to guide the code synthesis, where i denotes the instruction ID in the pre-collected dictionary, k denotes the program location, and x_i^k denotes whether the k -th instruction (in the synthesized code) holds ID i (in the dictionary). The code needs to satisfy a number of objective constraints, e.g., $\forall k, (\sum_{i \in ID} x_i^k) = 1$ which guarantees there is only one instruction placed at location k . We further encode the state changes (on register and memory objects) as transformations defined by x_i^k . The changes are constrained to be the same as the original code. Trigger injection is achieved by $(\sum_k x_t^k) \geq 1$ in which t is the ID of the trigger instruction, that is, the trigger instruction is at least inserted once. As such, we reduce the semantic-preserving trigger injection to a satisfiability modulo theories (SMT) solving task, where the trigger-injected code can be derived from a satisfying model of x_i^k , which essentially encodes the set of instructions placed at individual locations, including the trigger instruction and the needed patch instructions to make sure the semantics are preserved after injection.

Figure 5.11 presents the details. It first defines the inputs of the constraint generator. Λ_{reg} and Λ_{mem} are the register and memory stores collected from the randomized micro-executions. \mathbf{L}_n denotes the last program location of the subject block, i.e., $\Lambda_{reg/mem}(\mathbf{L}_{n+1})$ denotes the final program state. Ω and t denote the pre-defined instruction dictionary and the ID of trigger instruction (in the dictionary), respectively. The length of the generated block is pre-set as m . In practice, starting from the length of the original code (i.e., n), we gradually increase m until the synthesis succeeds or m reaches a predefined length ($n + 20$ in our setting).

Several auxiliary variables are introduced to help model the state changes. \mathbb{A} denotes the accessible memory objects (objects that have been accessed during micro-executions) which can be derived from the final memory store (i.e., $\Lambda_{mem}(\mathbf{L}_{n+1})$). Intuitively, semantic-equivalent blocks should access exactly the same memory objects. R_r^k and M_a^k denote the values of register r and memory object $[a]$, respectively, after executing the first k generated

Input:

Λ_{reg}	RegisterStore produced by the randomized evaluation
Λ_{mem}	MemoryStore produced by the randomized evaluation
\mathbf{L}_n	The last instruction's label of the subject block
Ω	Instruction candidates used during code synthesis ($\Omega ::= IDarrowInstruction$)
t	The ID of the trigger instruction, i.e., $\Omega(t)$ is the trigger instruction
m	The target length (i.e. # of instructions) of the generated code

To Solve:

x_i^k A boolean variable denotes whether the instruction (in Ω) with ID i is the k -th instruction in the generated code ($1 \leq k \leq m$, $i \in ID$, $x_i^k \in \{0, 1\}$)

Variables:

\mathbb{A}	Accessible memory objects, i.e., memory objects accessed by subject code ($\mathbb{A} = \text{dom}(\Lambda_{mem}(\mathbf{L}_{n+1}))$)
R_r^k	The value of register r after executing the first k generated instructions ($0 \leq k \leq m$, $r \in Register$)
M_a^k	The value stored in $[a]$ after executing the first k generated instructions ($0 \leq k \leq m$, $a \in \mathbb{A}$)
T^k	Whether the k -th generated instruction accesses non-accessible memory objects ($1 \leq k \leq m$, $T^k \in \{0, 1\}$)

Constraint Construction:

Step 1. Initialization

$$\begin{aligned} & \textcircled{1} \left(\bigwedge_{r \in \text{dom}(\Lambda_{reg}(\mathbf{L}_0))} R_r^0 = \Lambda_{reg}(\mathbf{L}_0)(r) \wedge \left(\bigwedge_{r \notin \text{dom}(\Lambda_{reg}(\mathbf{L}_0))} R_r^0 = \text{random}() \right) \right. \\ & \textcircled{2} \left. \bigwedge_{a \in \text{dom}(\Lambda_{mem}(\mathbf{L}_0))} M_a^0 = \Lambda_{mem}(\mathbf{L}_0)(a) \right) \end{aligned}$$

Step 2. Constructing Constraints by Induction

Assuming we have constructed the constraints for the first $k-1$ generated instructions, i.e., we have constructed R_r^{k-1} , M_a^{k-1} , and T^{k-1}

Temporary Variables:

${}^i R_r^k$: The value of register r after executing the first k generated instructions if the k -th one's ID is i

${}^i M_a^k$: The value stored in $[a]$ after executing the first k generated instructions if the k -th one's ID is i

${}^i T^k$: Whether the generated code accesses non-accessible memory after executing the first k generated instructions and the k -th one's ID is i

Construction Rules:

$$\begin{aligned} & \textcircled{3} \forall i \in ID, s.t. \Omega(i) \equiv r_d := r_1 \text{ op } r_2 \text{ arrow} \\ & \quad ({}^i R_{r_d}^k = R_{r_1}^{k-1} \text{ op } R_{r_2}^{k-1}) \wedge ({}^i T^k = 0) \wedge \left(\bigwedge_{r \neq r_d} {}^i R_r^k = R_r^{k-1} \right) \wedge \left(\bigwedge_{a \in \mathbb{A}} {}^i M_a^k = M_a^{k-1} \right) \\ & \star \textcircled{4} \forall i \in ID, s.t. \Omega(i) \equiv r_d := \mathbf{R}(r_a) \text{ arrow} \quad \text{if } a = R_{r_a}^{k-1} \notin \mathbb{A} \text{ then } {}^i T^k = 1; \\ & \quad \text{else } ({}^i R_{r_d}^k = M_a^{k-1}) \wedge ({}^i T^k = 0) \wedge \left(\bigwedge_{r \neq r_d} {}^i R_r^k = R_r^{k-1} \right) \wedge \left(\bigwedge_{a \in \mathbb{A}} {}^i M_a^k = M_a^{k-1} \right) \\ & \star \textcircled{5} \forall i \in ID, s.t. \Omega(i) \equiv \mathbf{W}(r_a, r_v) \text{ arrow} \quad \text{if } a = R_{r_a}^{k-1} \notin \mathbb{A} \text{ then } {}^i T^k = 1; \\ & \quad \text{else } ({}^i M_a^k = R_{r_v}^{k-1}) \wedge ({}^i T^k = 0) \wedge \left(\bigwedge_r {}^i R_r^k = R_r^{k-1} \right) \wedge \left(\bigwedge_{a' \neq a} {}^i M_{a'}^k = M_{a'}^{k-1} \right) \\ & \textcircled{6} \left(\bigwedge_r (R_r^k = \sum_{i \in ID} {}^i R_r^k \times x_i^k) \right) \wedge \left(\bigwedge_{a \in \mathbb{A}} (M_a^k = \sum_{i \in ID} {}^i M_a^k \times x_i^k) \right) \wedge (T^k = \sum_{i \in ID} {}^i T^k \times x_i^k) \end{aligned}$$

Step 3. Constructing the Objective Constraints

$$\begin{aligned} & \textcircled{7} \left(\bigwedge_{r \in \text{dom}(\Lambda_{reg}(\mathbf{L}_{n+1}))} R_r^m = \Lambda_{reg}(\mathbf{L}_{n+1})(r) \right) \wedge \left(\bigwedge_{r \notin \text{dom}(\Lambda_{reg}(\mathbf{L}_{n+1}))} R_r^m = R_r^0 \right) \\ & \textcircled{8} \bigwedge_{a \in \mathbb{A}} M_a^m = \Lambda_{mem}(\mathbf{L}_{n+1})(a) \quad \textcircled{9} \bigwedge_{1 \leq k \leq m} \sum_{i \in ID} x_i^k = 1 \quad \textcircled{10} \sum_{1 \leq k \leq m} T^k = 0 \\ & \textcircled{11} \sum_{1 \leq k \leq m} x_t^k \geq 1 \quad \textcircled{12} \bigwedge_{1 \leq k \leq m} \bigwedge_{i \in ID} (x_i^k = 1 \vee x_i^k = 0) \end{aligned}$$

\star Rules $\textcircled{4}$ and $\textcircled{5}$ can be encoded as a sequence of (consecutive/nested) *if-then-else* statements by enumerating all accessible addresses in \mathbb{A} , see running example 5.12

Figure 5.11. Rules of constraint construction

instructions. For instance, $R_{r_0}^0$ denotes the initial value of register r_0 , and M_{12}^2 denotes the value of [12] after executing the first two synthesized instructions. We introduce boolean variables T^k to denote whether the k -th generated instruction accesses any invalid memory object. Note that all T^k 's are constrained to be 0 during synthesis.

The constraints of keeping the same state changes are constructed by induction. At the initialization stage, we constrain all R_r^0 and M_a^0 . Specifically, R_r^0 is set as the initial value of r , if r has been initialized by the original code (constraint (1)). Otherwise, R_r^0 is randomly selected (constraint (1)) and we constrain register r to keep the same value in the final program state. This is reasonable as a code snippet accessing additional registers but preserving their initial values is considered semantic-equivalent in our context. M_a^0 is set as the initial value accordingly (constraint (2)). For now, memory behaviors are strictly constrained to be the same. Assuming stage $k-1$ is done properly, i.e., R_r^{k-1} , M_a^{k-1} , and T^{k-1} are well constrained, the next step is to construct R_r^k , M_a^k , and T^k . The overall idea is to enumerate all the instructions in the dictionary and model R_r^k , M_a^k , and T^k under the assumption of a specific instruction i being selected. This is done by introducing a few temporary variables ${}^iR_r^k$, ${}^iM_a^k$, and ${}^iT^k$. ${}^iR_r^k$ is defined as the value of R_r^k under the assumption that the k -th generated instruction is i , and ${}^iM_a^k$ and ${}^iT^k$ are defined in a similar fashion. Constraint (3) describes the construction rule for $r_d := r_1 \text{ op } r_2$. ${}^iR_{r_d}^k$ is updated as the outcome of the expression among $R_{r_1}^{k-1}$ and $R_{r_2}^{k-1}$, while other registers except r_d remain untouched, i.e., R_r^k inherits the value of R_r^{k-1} . ${}^iT^k$ stays false and all memory objects remain unchanged, as memory is not involved. Constraint (4) describes the rule for $r_d := \mathbf{R}(r_a)$. It first checks whether $[R_{r_a}^{k-1}] \equiv [a]$ is accessible. If not, ${}^iT^k$ is marked as true to indicate an access violation. Otherwise, ${}^iR_{r_d}^k$ and other objects are updated according to the semantics. Likewise, constraint (5) defines the rule of the memory write operations. Observe that constraints (4) and (5) are not typical bit-vector operations but can be encoded as several *if-then-else* statements in Z3. Interested readers can refer to our running example in Figure 5.12. After enumerating all ${}^iR_r^k$, ${}^iM_a^k$, and ${}^iT^k$, variables x_i^k are used to select the proper R_r^k , M_a^k , and T^k (constraint (6)). For instance, $R_{r_0}^k$ equals to $\sum_{i \in ID} {}^iR_{r_0}^k \times x_i^k$. Note that there is only one x_i^k (i.e., x_1^k) solved as 1 and the rest 0, so that $R_{r_0}^k$ is selected as ${}^1R_{r_0}^k$.

The *objective constraints* are listed at the bottom of the figure. Constraints (7) and (8) guarantee the synthesized code has the same outcomes. Note that any unused register r (by the original code) needs to keep its initial value (to avoid global side-effects). Constraints (9), (10), and (11) ensure that the generated block is legitimate, shares exactly the same memory behaviors, and has triggers injected, respectively. Note that as mentioned in Section 5.4.1, we can have a pool of trigger candidates after the trigger inversion. We do not explicitly select the trigger from the candidate pool, but instead let the solver decide which one to choose. Specifically, the constraint (11) originally guarantees a specific trigger instruction t is injected. When having a pool of trigger instructions, i.e. $P = \{t_1, t_2, \dots, t_p\}$, we change the constraint to $\sum_{t \in P, 1 \leq k \leq m} x_t^k \geq 1$, i.e., at least one trigger instruction in the pool being injected. The last constrain guarantees x_i^k has a boolean value.

Example Continued. We use Figure 5.12 to illustrate the constraint generation process. The inputs consist of the subject block from Table 5.1, its last label L_3 , the program state Λ analyzed from Table 5.1, an instruction dictionary Ω containing 7 instructions, a trigger instruction $r_0 := r_0 \times 2$, and a target length $m = 3$. At the initialization stage, $R_{r_0}^0$ is set as register r_0 's initial value 6 and $R_{r_1}^0$ is randomly initialized as 8. There are two accessible memory objects, [12] and [19]. M_{12}^0 and M_{19}^0 are then set as the corresponding initial values. During induction, assuming the stage $k-1$ is done, we start the construction of stage k . Recall that we need to enumerate all instructions and construct the corresponding ${}^iR_{r_0}^k$, ${}^iR_{r_1}^k$, ${}^iM_{12}^k$, ${}^iM_{19}^k$, and ${}^iT^k$. Instruction 0 : $r_0 := r_0 \times 2$ sets ${}^0R_{r_0}^k$ as ${}^0R_{r_0}^{k-1} \times 2$ and keeps ${}^0T^k$ false. Any unmentioned variable inherits the value of its ancestor, e.g., ${}^0R_{r_1}^k = {}^0R_{r_1}^{k-1}$, and we hence omit these constraints in the running example. Expression instructions 1 and 2 follow a similar rule to construct the variables. Instruction 3 : $r_0 := \mathbf{R}(r_1)$ uses a nested *if-then-else* statement to construct ${}^3R_{r_0}^k$ and ${}^3T^k$. Specifically, if the target address (i.e., $R_{r_1}^{k-1}$) equals to 12 or 19, ${}^3R_{r_0}^k$ is set as M_{12}^{k-1} or M_{19}^{k-1} , respectively. Otherwise, the memory read operation is invalid, and we set ${}^3R_{r_0}^k$ as $R_{r_0}^{k-1}$ (which is a meaningless placeholder) and flip ${}^3T^k$ as true to indicate the access violation. Instruction 4 is handled the same way. The next instruction 5 : $\mathbf{W}(r_0, r_1)$ also leverages *if-then-else* statements to describe the memory write operation. That is, the memory object ${}^5M_{12}^k$ (or ${}^5M_{19}^k$) is set as $R_{r_1}^{k-1}$ if the target address

Subject Block:

$$\mathbf{L}_0 : r_1 := r_0; \quad \mathbf{L}_1 : r_0 := r_0 + r_1; \quad \mathbf{L}_2 : r_1 := \mathbf{R}(r_0); \quad \mathbf{L}_3 : \mathbf{W}(r_1, r_0)$$

Input:

- (A) $\Lambda_{reg} \equiv \{L_0 \mapsto \{r_0 \mapsto 6\}, L_4 \mapsto \{r_0 \mapsto 6, r_1 \mapsto 19\}\}$ (D) $t = 0$, i.e., trigger instruction $\Omega(0) \equiv r_0 := r_0 \times 2$
 (B) $\Lambda_{mem} \equiv \{L_0 \mapsto \{12 \mapsto 19, 19 \mapsto 96\}, L_4 \mapsto \{12 \mapsto 19, 19 \mapsto 12\}\}$ (E) $L_n = L_3$ (F) $m = 3$
 (C) $\Omega \equiv \{0 \mapsto r_0 := r_0 \times 2, 1 \mapsto r_0 := r_0 + r_1, 2 \mapsto r_1 := r_0, 3 \mapsto r_0 := \mathbf{R}(r_1),$
 $4 \mapsto r_1 := \mathbf{R}(r_0), 5 \mapsto \mathbf{W}(r_0, r_1), 6 \mapsto \mathbf{W}(r_1, r_0)\}$

Variables:

- (G) $\mathbb{A} = \{12, 19\}$ (H) $R_r^k \in \{R_{r_0}^0, R_{r_0}^1, R_{r_0}^2, R_{r_0}^3, R_{r_1}^0, R_{r_1}^1, R_{r_1}^2, R_{r_1}^3\}$
 (I) $M_a^k \in \{M_{12}^0, M_{12}^1, M_{12}^2, M_{12}^3, M_{19}^0, M_{19}^1, M_{19}^2, M_{19}^3\}$ (J) $T^k \in \{T^0, T^1, T^2, T^3\}$

Initialization:

- (K) $(R_{r_0}^0 = 6) \wedge (R_{r_1}^0 = \text{random}()) = 8$ (L) $(M_{12}^0 = 19) \wedge (M_{19}^0 = 96)$

Induction:

Assuming stage $k-1$ is done, i.e., we have constructed R_r^{k-1} , M_a^{k-1} , and T^{k-1}

- (M) Constraints of ${}^i R_r^k$, ${}^i M_a^k$, and ${}^i T^k$: ${}^i R_r^k = {}^i R_r^{k-1}$, ${}^i M_a^k = {}^i M_a^{k-1}$ if not mentioned
if-then-else statement is denoted by $\mathbf{If}(\text{cond}, \text{then}, \text{else})$ in **Z3**

i	$\Omega(i)$	Constraints
0	$r_0 := r_0 \times 2$	$({}^0 R_{r_0}^k = R_{r_0}^{k-1} \times 2) \wedge ({}^0 T^k = 0)$
1	$r_0 := r_0 + r_1$	$({}^1 R_{r_0}^k = R_{r_0}^{k-1} + R_{r_1}^{k-1}) \wedge ({}^1 T^k = 0)$
2	$r_1 := r_1$	$({}^2 R_{r_1}^k = R_{r_0}^{k-1}) \wedge ({}^2 T^k = 0)$
3	$r_0 := \mathbf{R}(r_1)$	${}^3 R_{r_0}^k = \mathbf{If}(R_{r_1}^{k-1} = 12, M_{12}^{k-1}, \mathbf{If}(R_{r_1}^{k-1} = 19, M_{19}^{k-1}, R_{r_0}^{k-1}))$ ${}^3 T^k = \mathbf{If}(R_{r_1}^{k-1} = 12, 0, \mathbf{If}(R_{r_1}^{k-1} = 19, 0, 1))$
4	$r_1 := \mathbf{R}(r_0)$	${}^4 R_{r_1}^k = \mathbf{If}(R_{r_0}^{k-1} = 12, M_{12}^{k-1}, \mathbf{If}(R_{r_0}^{k-1} = 19, M_{19}^{k-1}, R_{r_1}^{k-1}))$ ${}^4 T^k = \mathbf{If}(R_{r_0}^{k-1} = 12, 0, \mathbf{If}(R_{r_0}^{k-1} = 19, 0, 1))$
5	$\mathbf{W}(r_0, r_1)$	${}^5 M_{12}^k = \mathbf{If}(R_{r_0}^{k-1} = 12, R_{r_1}^{k-1}, M_{12}^{k-1})$ ${}^5 M_{19}^k = \mathbf{If}(R_{r_0}^{k-1} = 19, R_{r_1}^{k-1}, M_{12}^{k-1})$ ${}^5 T^k = \mathbf{If}(R_{r_0}^{k-1} = 12, 0, \mathbf{If}(R_{r_0}^{k-1} = 19, 0, 1))$
6	$\mathbf{W}(r_1, r_0)$	${}^6 M_{12}^k = \mathbf{If}(R_{r_1}^{k-1} = 12, R_{r_0}^{k-1}, M_{12}^{k-1})$ ${}^6 M_{19}^k = \mathbf{If}(R_{r_1}^{k-1} = 19, R_{r_0}^{k-1}, M_{12}^{k-1})$ ${}^6 T^k = \mathbf{If}(R_{r_1}^{k-1} = 12, 0, \mathbf{If}(R_{r_1}^{k-1} = 19, 0, 1))$

$$(N) (R_{r_0}^k = \sum_{0 \leq i \leq 6} {}^i R_{r_0}^k \times x_i^k) \wedge (R_{r_1}^k = \sum_{0 \leq i \leq 6} {}^i R_{r_1}^k \times x_i^k) \wedge (M_{12}^k = \sum_{0 \leq i \leq 6} {}^i M_{12}^k \times x_i^k) \wedge (M_{19}^k = \sum_{0 \leq i \leq 6} {}^i M_{19}^k \times x_i^k) \wedge (T^k = \sum_{i \in ID} {}^i T^k \times x_i^k)$$

Objective Constraints:

$$(O) (R_{r_0}^3 = 6) \wedge (R_{r_1}^3 = 19) \wedge (M_{12}^3 = 19) \wedge (M_{19}^3 = 12) \quad (P) (\sum_{0 \leq i \leq 6} x_i^1 = 1) \wedge (\sum_{0 \leq i \leq 6} x_i^2 = 1) \wedge (\sum_{0 \leq i \leq 6} x_i^3 = 1)$$

$$(Q) \bigwedge_{1 \leq k \leq 3} \bigwedge_{0 \leq i \leq 6} (x_i^k = 1 \vee x_i^k = 0) \quad (R) x_0^1 + x_0^2 + x_0^3 \geq 1 \quad (S) T^1 + T^2 + T^3 = 0$$

Results:

$x_0^1 = x_4^2 = x_6^3 = 1$ and the rest are 0. The generated code s' is:

$$\mathbf{L}_0 : \Omega(0) \equiv r_0 := r_0 \times 2; \quad \mathbf{L}_1 : \Omega(4) \equiv r_1 := \mathbf{R}(r_0); \quad \mathbf{L}_2 : \Omega(6) \equiv \mathbf{W}(r_1, r_0)$$

Figure 5.12. Running example for constraint construction

equals to 12 (or 19), and remains unchanged otherwise. The rest constraints are easy to understand and the description is hence elided. A satisfying model of above mentioned constraints is that $x_0^1 = x_4^2 = x_6^3 = 1$ and the rest are 0, deriving a trigger-embedded block $\mathbf{L}_0 : \Omega(0) \equiv r_0 := r_0 \times 2; \quad \mathbf{L}_1 : \Omega(4) \equiv r_1 := \mathbf{R}(r_0); \quad \mathbf{L}_2 : \Omega(6) \equiv \mathbf{W}(r_1, r_0)$. Observe that it has the same semantics as the original code and the trigger injected. \square

5.6 Evaluation

We evaluate PELICAN on 5 binary analysis tasks and 15 models. The evaluation studies the effectiveness of PELICAN in inducing misclassification on subject models in both white-box and black-box scenarios. We investigate the underlying reason that backdoors exist in the pre-trained models. The experiments are performed on a server equipped with a 48-cores CPU (Intel(R) Xeon(R) Silver 4214 CPU @ 2.20GHz), 256G main memory, and 8 NVIDIA Quadro RTX 6000 GPUs.

5.6.1 Experiment Setup

Tasks and Models. We evaluate on 5 binary analysis tasks and 15 models, which are presented in Table 5.2. The first column denotes the tasks. The second and third columns show the binary analyses and their model architectures. The fourth and fifth columns present the evaluation metrics used for measuring the performance of the subject models and their values. The last column denotes how the models are obtained. In total, we have 10 Transformer-based models and 5 RNN-based models. The performance of all the evaluated models is consistent with that in the original papers. We chose to induce the misbehavior of the instruction boundary detection model (i.e., XDA-call) for a specific instruction type (i.e., call), to demonstrate a real-world attack that the attacker tries to hide a few critical instructions (e.g., calling a malicious function) instead of completely breaking the disassembler.

Attack Settings. We randomly select 10% binaries from our SPEC2000 dataset (around 2000 functions) to invert backdoor triggers, and launch attacks with samples from the author-provided test sets. This is similar to the real-world attack scenario where the attackers can prepare their own dataset for backdoor inversion without prior knowledge of the victim

Table 5.2. Summary of models used, along with how we collect these models (i.e., Source). \underline{P} , \underline{T} , and \underline{Q} denote the models are provided by the authors, trained with the author-provided dataset, and trained with our own dataset, respectively. The techniques named with a suffix ++ are enhanced by PalmTree [201], an instruction embedding technique.

Task	Technique	Architecture	Metric	Score	Source
Dis-assembly	BiRNN-func ³ [120]	Bidirectional RNN	Precision	99.12%	\underline{Q}
	XDA-func ³ [118]	Transformer	Precision	99.36%	\underline{P}
	XDA-call ⁴ [118]	Transformer	Precision	100.00%	\underline{P}
Function Signature Recovery	StateFormer [129]	Transformer	Precision	96.60%	\underline{T}
	EKLAVYA [133]	RNN	Precision	70.29%	\underline{T}
	EKLAVYA++ [133]	RNN + TE ¹	Precision	74.25%	\underline{T}
Function Name Prediction	in-nomine [135]	Transformer	Precision ²	33.97%	\underline{P}
	in-nomine++ [135]	Transformer + TE ¹	Precision ²	25.26%	\underline{T}
Compiler Provenance	S2V [123]	RNN + structure2vec	Precision	73.06%	\underline{T}
	S2V++ [123]	RNN + TE ¹ structure2vec	Precision	73.64%	\underline{T}
Binary Similarity	Trex [121]	Transformer	Top@1 Acc	91.11%	\underline{P}
	SAFE [122]	Bidirectional RNN	Top@1 Acc	89.29%	\underline{P}
	SAFE++ [122]	Bidirectional RNN + TE ¹	Top@1 Acc	87.01%	\underline{T}
	S2V-B [123]	RNN + structure2vec	Top@1 Acc	81.10%	\underline{T}
	S2V-B++ [123]	RNN + TE ¹ structure2vec	Top@1 Acc	82.78%	\underline{T}

¹ The instruction embedding is generated by a transformer model [201].

² We consider a prediction correct if there are more than 5% tokens correctly predicted.

³ BiRNN-func and XDA-func are to detect function boundaries.

⁴ XDA-call is to detect boundaries of all call instructions.

model’s training process. The epoch number and the learning rate of the trigger inversion are 50 and 0.1, respectively.

Computational Cost. As mentioned in Section 5.2, the memory usage of PELICAN’s trigger inversion is dominated by the $119640 \times 17 \times 768$ matrix, which consumes around 6G GPU memory. Note that multiplying the dictionary embedding matrix with the weight vector produces a small matrix of size 17×768 . We argue that it is within capacity of modern GPUs. And there are many attacks [202, 203] in CV and NLP domains that consume

much more resources but remain feasible for modern GPUs. On average, PELICAN’s trigger inversion takes 30 minutes to generate an effective backdoor for each model.

Attack for Disassembly Models. The disassembly models take bytes as input instead of assembly instructions. We hence develop a dedicated attack for these disassembly models (i.e., BiRNN-func, XDA-func, and XDA-call). Specifically, the backdoor triggers are inverted in form of bytes and injected at locations that cannot be reached during runtime, e.g., the preceding bytes before each function entrypoint.

Threats to Validity. In the context of attacking disassembly models, the semantic-preserving syntax-aware effort is not utilized, and the measurement of functionality preservation reflects the efficacy of the underlying binary rewriting engine rather than that of PELICAN. However, our intention in including disassembly models in our evaluations is to demonstrate the pervasiveness of backdoor vulnerability in various deep learning binary analysis tasks. It is possible that other adversarial machine learning techniques may achieve similar attack success rates as PELICAN on disassembly models.

5.6.2 Attack Effectiveness

The attack results of PELICAN are shown in Table 5.3. The first column presents the binary analysis techniques and the second column the original performance of these techniques. We apply PELICAN with different backdoor sizes, i.e., the number of instructions that can be injected to a binary function. Note that all the binary analysis techniques are originally evaluated at the function level. We hence follow the same setting by injecting the backdoor trigger in each function to induce misclassification. Columns 3-14 show the attack performance of PELICAN with different backdoor sizes. Column ASR denotes the attack success rate, i.e., the percentage of functions that a subject model produces correct predictions for before attack but wrong after. Column Score denotes the performance of the subject model measured using its original metric as shown in Table 5.2. Column Dis. presents the edit distance between the trigger-injected function and its original version. Note that the edit distance is presented as the ratio to the size of original functions. For example, with x the edit distance and y the original size, x/y is presented. Observe that with only one

Table 5.3. Attack effectiveness of PELICAN with different trigger sizes. **ASR** denotes the attack success rate and **Score** the model performance. **Dis.** denotes the edit distance between the trigger-injected functions and the original versions over the originals.

Technique	Original Score	Backdoor Size: 1			Backdoor Size: 3			Backdoor Size: 5			Backdoor Size: 7		
		ASR	Score	Dis.	ASR	Score	Dis.	ASR	Score	Dis.	ASR	Score	Dis.
BiRNN-func	99.12%	91.24%	8.50%	0.15%	96.35%	3.39%	0.46%	98.12%	1.62%	0.76%	98.12%	1.62%	1.06%
XDA-func	99.36%	93.44%	6.53%	0.15%	98.31%	1.68%	0.46%	98.32%	1.67%	0.76%	98.32%	1.67%	1.06%
XDA-call	100.00%	99.49%	0.51%	3.95%	99.57%	0.43%	6.59%	99.57%	0.43%	9.23%	99.57%	0.43%	11.86%
StateFormer	96.60%	45.09%	53.78%	11.73%	77.52%	21.73%	35.19%	89.51%	10.66%	58.65%	94.88%	4.89%	82.11%
EKLAVYA	70.29%	55.33%	32.91%	2.76%	84.43%	12.88%	7.16%	92.93%	6.45%	12.84%	96.11%	3.73%	16.15%
EKLAVYA++	74.25%	60.49%	31.85%	2.12%	89.63%	8.51%	6.36%	92.63%	8.05%	10.60%	93.81%	6.75%	14.83%
in-nomine	33.97%	42.85%	19.42%	2.44%	68.25%	10.79%	8.56%	83.75%	5.52%	15.89%	85.42%	4.95%	19.55%
in-nomine++	25.26%	47.26%	13.32%	1.83%	81.52%	4.67%	6.72%	87.65%	3.12%	11.61%	92.25%	1.96%	17.72%
S2V	73.06%	42.64%	42.64%	5.90%	73.73%	19.51%	17.71%	83.66%	12.12%	29.52%	89.66%	7.64%	41.33%
S2V++	73.64%	32.87%	51.71%	4.78%	73.68%	19.91%	14.35%	85.28%	11.06%	23.92%	90.88%	6.81%	33.48%
Trex	91.11%	59.32%	37.83%	1.74%	89.89%	9.50%	5.22%	96.40%	3.39%	8.70%	98.30%	1.60%	12.18%
SAFE	89.29%	74.18%	23.68%	5.09%	94.44%	5.20%	15.26%	98.04%	1.84%	27.98%	98.99%	0.96%	38.15%
SAFE++	87.01%	64.75%	31.58%	3.82%	94.76%	4.98%	11.45%	98.79%	1.15%	19.08%	99.71%	0.30%	26.71%
S2V-B	81.10%	89.55%	8.86%	4.52%	96.58%	3.09%	13.57%	98.14%	1.66%	22.62%	98.94%	1.02%	31.67%
S2V-B++	82.78%	59.36%	34.52%	6.03%	78.68%	18.19%	18.09%	86.12%	11.93%	30.16%	89.97%	8.79%	42.22%
Average	78.46%	63.87%	26.51%	3.80%	86.49%	9.63%	11.14%	92.59%	5.38%	18.82%	95.00%	3.54%	26.01%

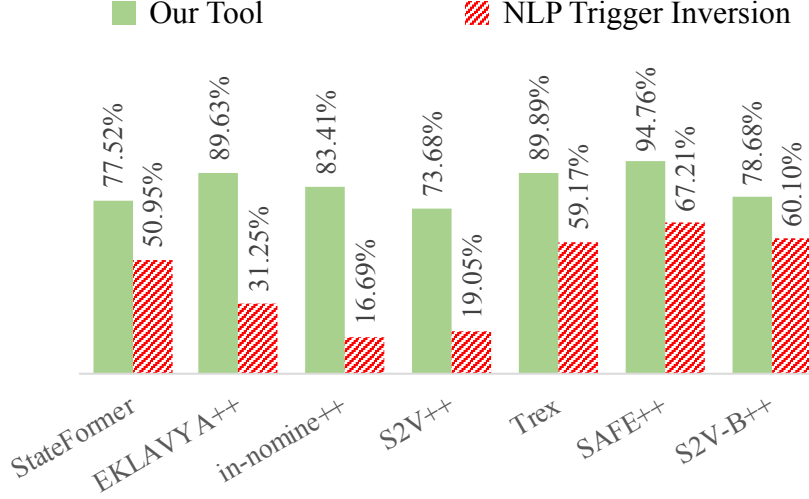


Figure 5.13. Comparison between an adopted NLP trigger inversion technique and PELICAN

injected instruction, PELICAN has already over 90% ASR on BiRNN-func, XDA-func, and XDA-call. The ASR on other models such as EKLAVYA++, SAFE, SAFE++, and S2V-B are also reasonable, with an over 60% ASR when only one instruction is injected. When the backdoor size is increased to 5, PELICAN is able to break all the evaluated models with over 80% ASR. The performance of these models measured by their corresponding metrics is only 5.38% on average. The edit distance increases when injecting more backdoor instructions. This is reasonable as PELICAN needs to add more instructions in each function. One may notice that the edit distance is relatively high for StateFormer. It is because the functions from StateFormer’s test set are shorter than other, rendering the ratio high. We find when the backdoor size is 3, PELICAN achieves a good balance with 86.49% ASR and 11.14% edit distance on average.

5.6.3 Comparison with Baselines

We compare PELICAN with three baselines: using an adopted NLP inversion technique to invert triggers, using opaque predicates to inject triggers, and a state-of-the-art per-instance adversarial attack that does not rely on trigger inversion.

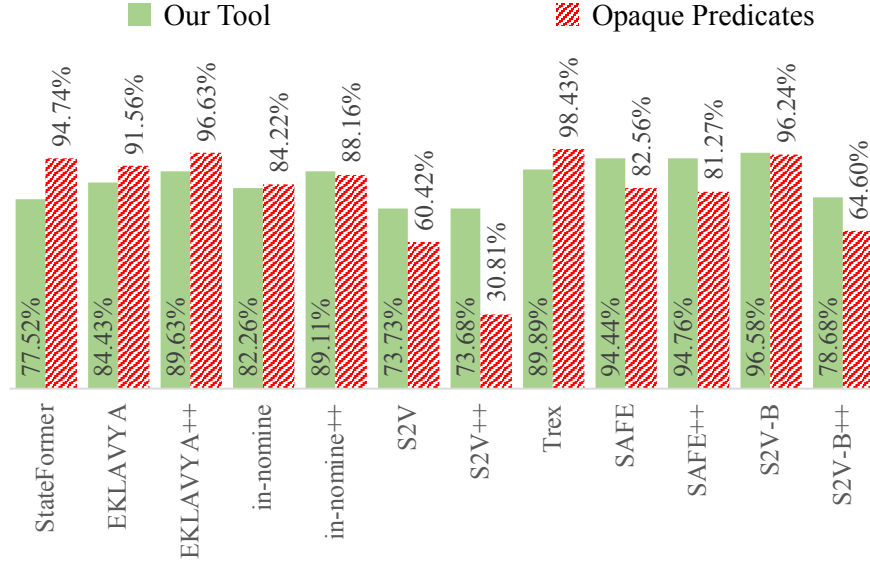


Figure 5.14. Comparison between a baseline method using opaque predicts to inject triggers and PELICAN

Adopted NLP Trigger Inversion Technique. The baseline method adopts an existing NLP trigger inversion technique [183]. During the trigger inversion, we gradually discard the opcode tokens that take more operands, so that the baseline method can eventually invert a legitimate one. For instance, in the worst case, only nullary opcodes are available and any inverted token is a legitimate nullary instruction. We use all the 7 models that take tokenized assembly sequence as input, and compare the baseline method’s attack performance to PELICAN’s. The backdoor size is 3 and the result is shown in Figure 5.13. Observe that the baseline method has at most 67.21% ASR on SAFE++, whereas PELICAN has 94.76% ASR. The baseline performs worst on in-nomine++ with only 16.69%. PELICAN, on the other hand, still has 81.52% ASR. This is because the target triggers, i.e., the ones that the models are undesirably overfitting to, are hard to invert when the inversion technique does not take the assembly syntax into account. Overall, PELICAN substantially outperforms the baseline inversion method.

Trigger Injection by Opaque Predicates. Another baseline method leverages opaque predicates to inject backdoor triggers [204]. It aims to attack Android malware classifiers



Figure 5.15. The runtime trigger coverage

based on code features, e.g., code size. It uses opaque predicates to inject arbitrary bytes to subject malwares. Opaque predicates are predicates whose true branches can never be taken. As such, any instructions (or even arbitrary bytes) guarded by these predicates will never be executed. Since our models are not based on code size, their technique is not directly applicable. We hence leverage its opaque predicate transformation to inject the trigger instructions generated by our inversion technique. We compare attack performance on the 12 models that are not the disassembly ones. We additionally study the impact of data sanitization (e.g., remove instructions that are not covered during execution) and the runtime overhead of trigger-injected binaries.

Attack Success Rate. Figure 5.14 presents the ASRs of the two methods. PELICAN achieves comparable ASRs with the baseline methods on all the models. Observe that the baseline attack performs slightly better for function signature recovering tasks (i.e., StateFormer, EKLAVYA, and EKLAVYA++). This is because it always just injects triggers at the beginning of functions, achieving the maximum attack effects on function signature recovery models. However, this may backfire. Observe that the baseline has only 30.81% ASR on

S2V++, while PELICAN’s still reaches 73.73%. Overall, compared to the baseline, our attack is just as effective and more stable.

Input Sanitization. Many existing research works [204–207] have emphasized the importance of inconspicuousness and hence aim at generating stealthy backdoored/adversarial samples. Malware Makeover [205], a state-of-the-art malware evasion technique, further asserts the possible defence of pruning out crafted bytes in unreachable regions of the binary. De-obfuscation techniques [208–210] are also largely adapted by the RE community. We hence study possible defense by sanitizing input binaries. One possible approach is to dynamically execute the program and eliminate the uncovered code. Note that although one cannot discard unexecuted code in general, in the context of de-obfuscation (before passing an executable to a malware classifier), it is justifiable to suppress the un-executed instructions [211, 212]. We inject different triggers (which are inverted from 12 subject models) into the SPEC2000 programs and study the trigger coverage (i.e., the percentage of dynamically covered trigger instructions). We use the reference input set provided by SPEC2000 to collect runtime information. Note that we cannot collect the trigger coverage for the author-provided test programs due to the lack of valid input data. Figure 5.15 details the coverage. Observe that even in the worst case, PELICAN still achieves 87.12% trigger coverage, i.e., almost all triggers injected by PELICAN are dynamically executed (and hence an integral part of the sanitized programs). On the other hand, triggers injected by the baseline method can be easily eliminated since the dead code is never executed.

We additionally implement a naïve opaque predicate detector inspired by [191]. It is a hybrid static-dynamic technique without considering complex program semantics. The results show 54.84% opaque predicates are detected, indicating that more than half of injected triggers can be eliminated. Many other state-of-the-art opaque predicate detection techniques [185–190, 213] are believed to have better performance.

Runtime Overhead. We use SPEC2000 programs to study the runtime overhead of backdoor-injected binaries by different trigger injection methods. To avoid the randomness from each execution, we run all the binaries for 3 times and obtain the average. On average, the binaries with PELICAN’s injected backdoors have 4.36%, 8.21%, 10.98%, and 15.13% runtime overhead when the trigger size is 1, 3, 5, and 7, respectively. The baseline opaque

predicates, on the other hand, has around 200% runtime overhead, rendering the attack infeasible for performance-sensitive applications. Recall that we adapt the settings of an existing work [204], where opaque predicates are to validate if a set of random values satisfy a preset 3-SAT formula, to avoid being easily determined as a bogus predicate (by a static analysis). The runtime overhead is mainly caused by the execution of opaque predicates [168]. Detailed results are provided in Table 5.5 (in Section 5.6.6).

Instance-specific Attack. Malware Makeover [205] (hereinafter referred to as *MalMakeover*) is a state-of-the-art per-instance attack against malware classifiers. For a given malware sample, MalMakeover iteratively applies semantics-preserving transformations upon the sample until the resultant variant induces misclassification. To efficiently guide the transformations, MalMakeover proposes an optimization algorithm which, at each iteration, only selects a transformation that can entail a lower attack *CW loss* [157] (compared to the current malware variant). MalMakeover achieves a high evasion rate against DL-based malware detectors [205]. It is hence interesting to compare its performance on binary analysis models with ours.

Experiment Configuration. In our modifications to the original version of MalMakeover, we have made two significant alterations. Firstly, we have reduced the scope of mutation from an entire program to a single function. In contrast to the original MalMakeover, which selected functions to mutate in an iterative manner, our variant focuses exclusively on a given function. This modification was necessary as our subject models accept input in the form of a single function, as opposed to the original MalMakeover, which took malware binaries as input. Secondly, we have replaced the disassembling frontend of MalMakeover with a custom implementation that accepts text-form assembly code as input. This change was required as the test sets provided by the authors of the subject models did not consist of complete programs, but rather individual functions. To accomplish this, we utilized Keystone for assembly and Capstone for disassembly, which provided the necessary information at the instruction-level. Additionally, since the original MalMakeover relied on a register liveness analysis, which is not supported by Capstone, we developed such an analysis ourselves. Our variant of MalMakeover supports both in-place randomization and displacement.

Table 5.4. Attack success rates of untargeted attacks

Model	Tool	Backdoor Size			
		1	3	5	7
EKLAVYA	PELICAN	55.33%	84.43%	92.93%	96.11%
	MalMakeover	29.07%	57.31%	70.34%	76.64%
EKLAVYA++	PELICAN	60.49%	89.63%	92.63%	93.81%
	MalMakeover	25.82%	58.13%	67.80%	75.61%
in_nomine	PELICAN	42.85%	68.25%	83.75%	85.42%
	MalMakeover	41.15%	67.85%	83.38%	85.37%
in_nomine++	PELICAN	47.26%	81.52%	87.65%	92.25%
	MalMakeover	41.81%	85.19%	87.18%	91.22%
StateFormer	PELICAN	45.09%	77.52%	89.51%	94.88%
	MalMakeover	50.82%	84.30%	84.34%	84.41%

It is worth noting that the original design of MalMakeover was intended for models that accept a program as input and output a predicted label, which is essential for the use of CW loss. However, for the disassembly and binary similarity tasks, the output is in the form of sequences and embedding vectors, respectively, making them incompatible with the original design of MalMakeover. Furthermore, MalMakeover calculates the difference of input embeddings before and after mutation. The S2V and S2V++ models, which take CFG-like graphs as inputs, pose a challenge in this regard as the embedding difference cannot be easily calculated after displacement mutation, which significantly alters the CFG. Consequently, we exclude disassembly models, binary similarity models, S2V, and S2V++ from our evaluation. In our study, we examine the efficacy of untargeted attacks (i.e., causing the model to misclassify a sample to any other label), the performance of targeted attacks (i.e., causing the model to misclassify a sample to a specified label), and the runtime overhead of mutated binaries.

Untargeted Attack. Table 5.4 presents the ASRs of untargeted attacks performed by PELICAN and MalMakeover. The first two columns denote the subject models and the attack techniques, respectively. Columns 3-6 present the ASRs with backdoor sizes of 1, 3, 5, and 7. It is important to note that MalMakeover does not reverse trigger instructions, thus the metric of backdoor size is not directly relevant. To make a meaningful comparison, for each

function sample, we guarantee that the edit distances of the function variants generated by PELICAN and MalMakeover are equivalent with respect to the original function. To calculate the edit distance, we compare the original assembly sequence of the subject function to the mutated sequence and consider each instruction as a unit for the purposes of counting. Observe that PELICAN and MalMakeover achieve comparable ASRs for untargeted attacks. Specifically, MalMakeover slightly outperforms PELICAN on `in_nomine++` with a backdoor size of 3 and on `StateFormer` with 1 and 3, while PELICAN achieves superior ASRs in the other settings. Also note that for `StateFormer`, although MalMakeover gets better ASRs with backdoor sizes of 1 and 3, its attack performance reaches an upper bound ASR of 85% when the backdoor size is larger than 3, while the ASR of PELICAN is close to 95% with a backdoor size of 7. We have also conducted a comprehensive evaluation of the average time required by PELICAN and MalMakeover to launch a successful untargeted attack. The parameters for backdoor size and the number of micro-execution instances were set to 5 and 3, respectively. Our results demonstrate that PELICAN requires an average of 19.23 seconds per function to achieve a successful untargeted attack, whereas MalMakeover requires only 7.68 seconds. It is worth noting that PELICAN necessitates an additional 30 minutes to generate an effective backdoor for each model through trigger inversion. We have carried out further investigation to understand the factors contributing to MalMakeover’s efficient performance in this regard. We observe that code displacement [214] is the most effective transformation. It moves a branching-free code piece to a new executable section and fills the original place by a leading `jmp` instruction (to redirect the control flow to the displaced code) and a set of *semantic-nop* instructions (i.e., instructions that cumulatively do not affect the memory or register values and have no side effects). This is mainly because, when pre-processing data, these binary analysis models do not take control-flow information into consideration. Instead, they assume all instructions of a function are contiguous in memory, and therefore consecutively collect instructions from the function entrypoint, until the first instruction that belongs to another function. Despite being valid in most cases, the contiguity assumption is broken by code displacement, where several instructions are displaced to a distinct memory space. As a result, these binary models can only access non-displaced instructions and understandably have an inferior performance. The success of MalMakeover

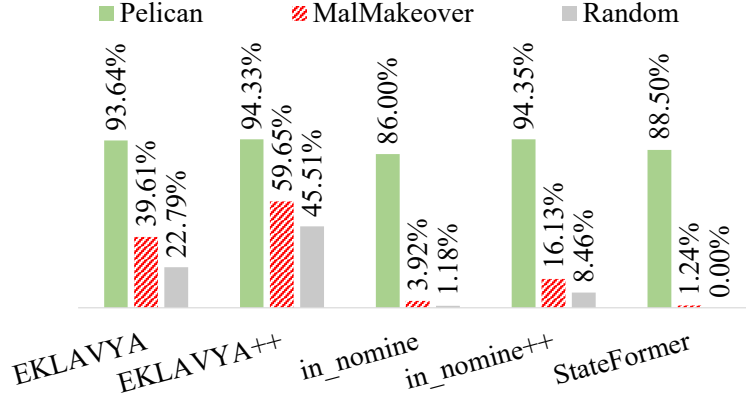


Figure 5.16. Attack success rates of targeted attacks

in untargeted attacks also suggests *the de-facto necessity of a proper data pre-processing step when developing binary analysis models, e.g., restoring displaced instructions.*

Targeted Attack. Figure 5.16 depicts the performance of targeted attacks against 5 models. The green, red, and grey bars denote the ASRs of PELICAN, MalMakeover, and a baseline approach that randomly selects backdoor instructions to inject without trigger inversion, respectively. The target label of EKLAVYA, EKLAVYA++, and StateFormer is “int”, and that of in_nomine and in_nomine++ is “init”. We adapt the setting of backdoor size 7. Observe that the best ASR MalMakeover can achieve is close to 60% and its ASRs on in_nomine and StateFormer are 3.92% and 1.24%, respectively. In comparison, PELICAN always achieves an ASR above 85% for all the subject models. The baseline method performs worst. We further investigate the underlying reason of the superiority of PELICAN in targeted attacks. As mentioned in Section 5.2, many models undesirably learn some low-level syntactic features. For example, StateFormer overfits on “add [r8], esi” and always predicts a function’s first argument as “int” as long as the function contains that add instruction (i.e., the trigger instruction). Note that PELICAN can effectively identify trigger instructions via the syntax-aware trigger inversion. Meanwhile, PELICAN’s semantic-preserving trigger injection is able to inject arbitrary inverted trigger instructions into the subject binary. On the other hand, MalMakeover mutates binaries by performing a set of pre-defined semantics-preserving transformations. These pre-defined transformations can introduce a few typical

types of instructions (e.g., `nop`, `push`, and `pop`) into the subject binary, while those in the trigger are beyond this scope. For example, it is less likely for these transformations to precisely produce the `"add [r8], esi"` instruction, rendering a suboptimal ASR of MalMakeover on StateFormer.

5.6.4 Functionality Preservation

PELICAN employs a semantic-preserving trigger injection technique to ensure that the functionality of the mutated binaries is retained. In this study, we have conducted an empirical examination of the preservation of functionality in backdoor-injected binaries. Our dataset consists of binaries from SPEC2000 [20], SPEC2006 [215], Binutils 2.39 [216], and Coreutils 8.25 [69]. These datasets are well-suited for our purposes as they come equipped with a large number of comprehensive test cases. For each binary, we have applied 12 backdoor triggers (inverted from 12 non-disassembly models), resulting in a total of 1800 mutated binaries. Our attacks were executed under two distinct scenarios: with and without access to the source code. In the former scenario, PELICAN was integrated into the compilation toolchain and inserted backdoor instructions into the compiler-generated assembly code, which was then converted into binary form by the default assembler. In the latter scenario, the subject binaries were first disassembled into reassembleable assembly code using datalog disassembly [64] and then instrumented by PELICAN. The results show that, in the source-assisted setting, all the mutated binaries produce the expected outputs on the benchmark test cases, demonstrating the effectiveness of PELICAN’s semantic-preserving trigger injection. In the binary-only setting, 93.3% of the mutated binaries produce the expected outputs, while the rest of them crash or produce incorrect outputs. These failures are due to limitations in the datalog disassembly process.

5.6.5 Why Backdoors Exist in These Models?

In this section, we investigate the underlying reason that backdoors exist in the models of three classification tasks (i.e., function signature recovery, function name prediction, and compiler provenance). The binary similarity models are not used as their outputs are embed-

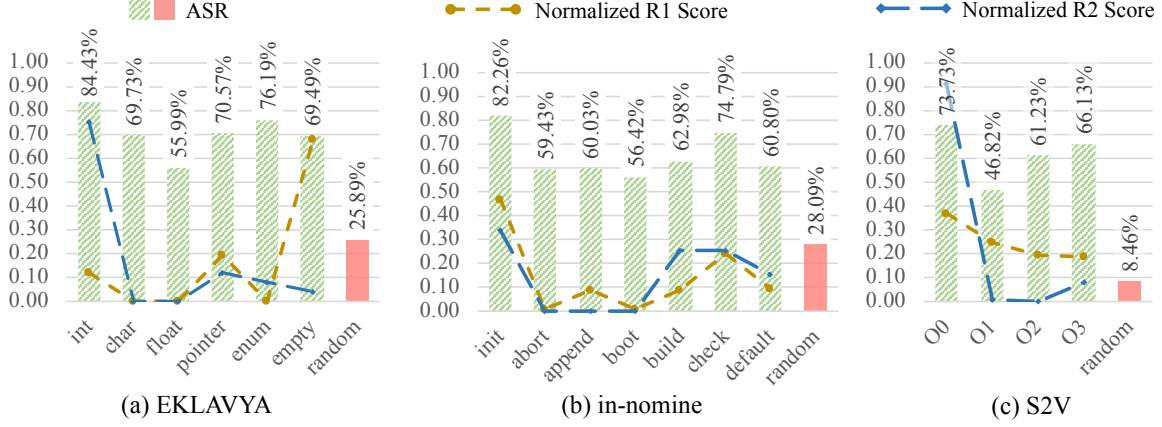


Figure 5.17. The relation between ASR and the underlying training bias. R1 and R2 score denote the sample-level and feature-level bias, respectively. *Random* denotes a baseline method that randomly selects trigger instructions.

ding vectors instead of some specific labels. For each task, we select two models and a few classes. Particularly, we study the relation between the attack success rate and the training bias evaluated by two metrics: *sample-level bias* and *feature-level bias*. The sample-level bias (R1) calculates the ratio of target class samples in the whole training set. The feature-level bias (R2) measures the ratio between two computed percentages: the percentage of samples containing backdoor instructions in the target class, and the percentage of samples containing backdoor instructions in other classes. For instance, assuming there are n training samples, n_a class-A samples, k_1 class-A samples containing the inverted trigger instructions, and k_2 non-class-A samples containing the triggers, $R1 = \frac{n_a}{n}$ and $R2 = \frac{k_1}{n_a} / \frac{k_2}{n - n_a}$. The results are shown in Figure 5.17. The x-axis denotes different target classes where *random* denotes a baseline method that the trigger instruction is randomly selected rather than inverted by PELICAN. The y-axis presents the ratio values. The bars show the ASRs on the test sets when using the corresponding class as the target during trigger generation. The dashed brown and blue lines show the results for the sample-level bias (R1) and the feature-level bias (R2), respectively. We have normalized the R1 and R2 scores for better visualization. In Figure 5.17(a), observe that the R2 line has a similar trend as the ASR bars, indicating EKLAVYA overfits on a few distinctive instructions in each class. Specifically, for the target

class `int`, both ASR and R2 are high, meaning EKLAVYA largely relies on the backdoor instructions (e.g., `mov ebx, edx`) for predicting class `int`. Similar observations can be made for other models as shown in Figure 5.17(b)(c)(d)(e)(f). The R1 line has the same trend but the trend is less significant, indicating a smaller contribution of the sample-level bias. To better understand the root cause of the vulnerabilities, we investigate a concrete case of EKLAVYA. Recall that the trigger instruction of “`mov ebx, edx`” is able to misguide EKLAVYA to incorrectly predict the subject function’s third argument as an integer. According to the x64 calling conventions [184], register `rdx` (i.e., the 64-bit extension of `edx`) is used to pass the third argument, which is however call-clobbered. Meanwhile, register `rbx` (i.e., the 64-bit extension of `ebx`) is a call-preserved general purpose register. To consistently use the argument’s value, mainstream compilers tend to load it into a call-preserved register. As a result, “`mov ebx, edx`” is frequently used by functions whose third arguments are 32-bit integers. It hence introduces natural bias into the training dataset, where such “`mov`” instructions are prevalent in the aforementioned functions but relatively rare in the rest of the dataset. Without the awareness and a proper remediation of such a bias, EKLAVYA undesirably relies on the low-level syntactic features (i.e., the present of “`mov ebx, edx`”) to make prediction, regardless of the underlying program semantics. Also observe that the baseline attack (i.e. *random*) performs poorly, rendering the importance of the syntax-aware trigger inversion. Overall, this study suggests that these models may focus on a few very distinctive instructions for prediction instead of relying on input semantics. PELICAN can hence diagnose and exploit the vulnerability by inverting the trigger instructions and injecting the triggers back to the subject functions.

5.6.6 Runtime Overhead

We use SPEC2000 programs to study the runtime overtime of backdoor-injected binaries by different trigger injection methods. To avoid the randomness from each execution, we run all the binaries for 3 times and obtain the average. The results are shown in Table 5.5. The first two columns denote the trigger injection methods and the programs. The following columns present the runtime overhead of these binaries when backdoors with different

Table 5.5. Runtime overhead

Injection	Program	Backdoor Size			
		1	3	5	7
PELICAN	164.gzip	4.16%	7.34%	6.36%	6.98%
	175.vpr	8.51%	12.48%	9.31%	14.85%
	176.gcc	1.56%	8.95%	8.17%	9.34%
	181.mcf	-0.80%	2.39%	0.80%	3.59%
	186.crafty	8.33%	9.52%	11.11%	23.02%
	197.parser	3.97%	8.85%	13.17%	14.64%
	252.eon	-1.96%	1.31%	11.11%	24.84%
	253.perlbmk	5.36%	4.46%	13.39%	20.76%
	254.gap	5.03%	9.75%	16.04%	15.09%
	255.vortex	10.56%	16.67%	26.76%	26.06%
	256.bzip2	4.25%	9.81%	10.25%	11.27%
	300.twolf	3.39%	7.00%	5.30%	11.17%
	Average	4.36%	8.21%	10.98%	15.13%
Opaque Predicate [204]	164.gzip	65.24%	66.46%	65.24%	64.01%
	175.vpr	79.41%	85.94%	84.55%	72.48%
	176.gcc	189.49%	190.66%	194.94%	179.38%
	181.mcf	10.56%	5.98%	1.79%	5.18%
	186.crafty	278.57%	271.03%	264.29%	260.71%
	197.parser	233.71%	233.71%	240.52%	217.82%
	252.eon	337.91%	341.18%	334.64%	321.57%
	253.perlbmk	248.21%	250.45%	259.38%	243.75%
	254.gap	396.86%	390.57%	384.28%	374.84%
	255.vortex	479.81%	453.99%	561.97%	449.30%
	256.bzip2	125.48%	124.01%	113.76%	110.83%
	300.twolf	85.10%	82.84%	77.20%	76.07%
	Average	210.86%	208.07%	215.21%	198.00%
MalMakeover [205]	164.gzip	1.96%	-0.37%	3.92%	6.61%
	175.vpr	4.95%	0.99%	9.31%	11.49%
	176.gcc	6.23%	2.72%	10.12%	22.18%
	181.mcf	-0.60%	-1.59%	-1.00%	-1.39%
	186.crafty	6.35%	11.90%	16.67%	33.73%
	197.parser	7.15%	9.76%	13.51%	29.40%
	252.eon	0.98%	22.22%	22.55%	50.65%
	253.perlbmk	15.85%	12.72%	20.76%	42.63%
	254.gap	6.60%	18.55%	18.24%	42.45%
	255.vortex	5.63%	20.89%	23.24%	66.90%
	256.bzip2	1.90%	1.46%	7.03%	17.57%
	300.twolf	1.47%	1.02%	1.81%	6.66%
	Average	4.87%	8.36%	12.18%	27.41%

sizes are injected. Obverse that binaries with PELICAN’s injected backdoors have low run-

Table 5.6. Transfer attack. **TS** denotes whether the victim model and surrogate model are trained from the same dataset. **V-ARS** and **T-ASR** denote the original attack success rate and the one for transfer attack, respectively.

TS	Victim Model	V-ASR	Surrogate Model	T-ASR	Degradation
Same Dataset	EKLAVYA	84.43%	EKLAVYA++	83.29%	1.14%
	EKLAVYA++	89.63%	EKLAVYA	88.55%	1.08%
	in-nomine	68.25%	in-nomine++	41.30%	26.95%
	in-nomine++	81.52%	in-nomine	77.72%	3.80%
	S2V	73.73%	S2V++	46.90%	26.83%
	S2V++	73.68%	S2V	44.70%	28.98%
	SAFE	94.44%	SAFE++	87.04%	7.40%
	SAFE++	94.76%	SAFE	88.54%	6.22%
	S2V-B	96.58%	S2V-B++	86.52%	10.06%
	S2V-B++	78.68%	S2V-B	75.84%	2.84%
Average					11.53%
Different Datasets	BiRNN-func	96.35%	XDA-func	95.33%	1.02%
	XDA-func	98.31%	BiRNN-func	97.96%	0.35%
	StateFormer	77.52%	EKLAVYA	38.43%	39.09%
	EKLAVYA	84.43%	StateFormer	57.22%	27.21%
	EKLAVYA++	89.63%	StateFormer	40.34%	49.29%
	Trex	89.89%	SAFE	86.69%	3.20%
	SAFE	94.44%	Trex	65.39%	29.05%
	SAFE++	94.76%	Trex	81.28%	13.48%
	S2V-B	96.58%	SAFE	84.44%	12.14%
	S2V-B++	78.68%	SAFE	73.46%	5.22%
Average					18.01%

time overhead (4.36-15.13% on average). MalMakeover achieves similar comparable runtime overhead. The baseline opaque predicates, on the other hand, has around 200% runtime overhead, rendering the attack infeasible for performance-sensitive applications.

5.6.7 Transfer Attack

The backdoors generated by PELICAN are effective in inducing misclassification on the evaluated models as discussed in previous sections. In this section, we study how the generated backdoor on one subject model can transfer to other models. This is particularly interesting as in the black-box attack scenario the subject model is inaccessible by the adversary. We call the model used in generating the backdoor the *surrogate model*, and the

model being attacked the *victim model*. We study two settings that the victim model and the surrogate model are trained from (1) the same dataset and (2) different datasets. Table 5.6 presents the results. Column TS denotes the aforementioned two settings. Column Victim Model denotes the victim model and column V-ASR shows the attack success rate of directly attacking the victim model (assuming accessibility). Column Surrogate Model denotes the surrogate model that the adversary utilizes. Column T-ASR shows the attack success rate on the victim model when using the backdoors generated on the surrogate model. Column Degradation denotes the ASR difference between the third and fifth columns. In the same dataset setting, PELICAN achieves more than 80% T-ASR for half of the models, demonstrating the potential transferability of PELICAN’s attack. The difference between V-ASR and T-ASR is only 11.53% on average, delineating the effectiveness of PELICAN in exposing the fundamental vulnerabilities of these models. In the different datasets setting, we have similar observations. Particularly, the backdoors generated by PELICAN on XDA-func and BiRNN-func can even achieve more than 95% T-ASR on the victim models. This is because the models undesirably learn the low-level syntactic features, e.g., function prologue generated by mainstream compilers, regardless of the training data. One may notice that the ASR degradation is as high as 50% for EKLAVYA++. It is reasonable given the completely different model architectures and training sets. We argue that a black-box attack with a worst case of 40.32% ASR poses realistic threat. On average, the attack success rate difference is 18.01% in the second setting, slightly larger than the previous one. This study demonstrates the possibility of using PELICAN to launch attacks on closed-source tools, which is discussed in Section 5.7.

5.7 Case Study

In this section, we present a case study of a black-box attack in which we launch transfer attacks on proprietary binary analysis models. Specifically, we investigate the security of DeepDi [119], a recently-proposed commercial disassembler. Our results have been disclosed in a responsible manner, and the authors of DeepDi have acknowledged the issue and are working to improve their product.

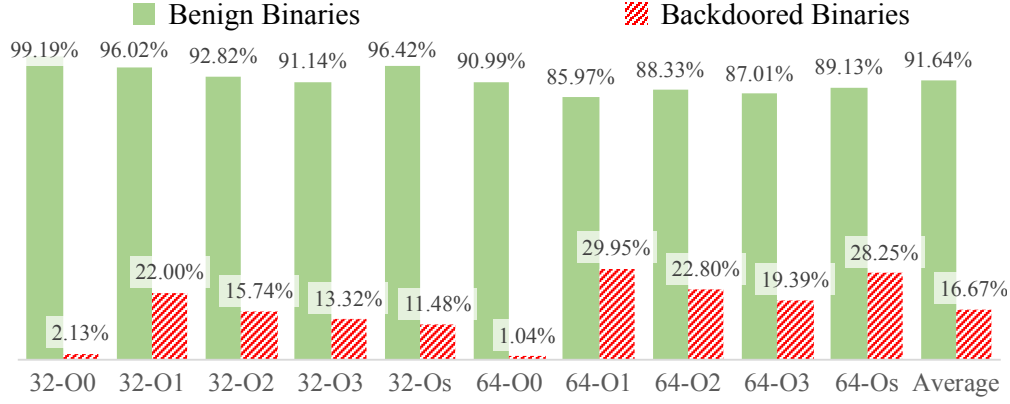


Figure 5.18. The F1 score for function boundary identification achieved by DeepDi [119], a closed-source commercial disassembler, is noteworthy.

DeepDi is a state-of-the-art GNN-based disassembler, proposed by DeepBits Technology [217]. It achieves low false positive and negative rates on both normal and obfuscated code. In this case study, the backdoor trigger is generated from XDA and the subject binaries are from SPEC2000. The trigger comprises 3 bytes and is injected as padding bytes before each function entrypoint. Figure 5.18 presents DeepDi’s F1 score for function boundary identification. The green and red bars denote benign and backdoor binaries, respectively. The x-axis denotes the compilation flags, and the last two columns show the average numbers. The y-axis denotes the F1 score. Observe that our attack is most effective for binaries compiled by O0, where the F1 scores have decreased from 99.19% to 2.13% and from 90.99% to 1.04% for 32-bit and 64-bit programs, respectively. On average, the F1 scores drop from 91.64% to 16.67% after the backdoor attack, demonstrating the effectiveness of black-box attack. Note that DeepDi, as a GNN-based model, is originally not within the scope of PELICAN (i.e., sequence models). However, the hypothesis of models overfitting on compiler-generated syntactic patterns holds, which enables such a transfer attack towards DeepDi. Further investigation shows that, the trigger, i.e., "85 e6 74", can be decoded as "test esi, esp; je XX" if followed by an arbitrary byte "XX". Note that test and je constitute a common code pattern of encoding conditional statements, inducing the misbehavior of DeepDi.

5.8 Summary

We study the security of DL models used in binary code analyses, which have a lot of downstream security applications. Our attack features a trigger generation technique for these models that produces instructions causing the models to misclassify, and a trigger injection technique that can preserve input program semantics.

6. DISCUSSION

The application of our proposed probabilistic analysis techniques is not confined to the specific binary analysis tasks discussed earlier. These methods offer a broad, flexible framework that can be employed in a wide array of applications where there is a need to deal with uncertainty. This is especially true for tasks that involve the extraction of higher-level insights from lower-level data.

In this chapter, we discuss the potential applicability of our probabilistic analysis across various fields of software engineering and security. The inherent adaptability of probabilistic analysis allows it to be used in diverse and complex scenarios, with the ability to navigate through the challenges posed by uncertainty and recover valuable insights from intricate low-level data. Specifically, we see significant potential for our proposed techniques for *disassembly*, *network protocol reverse engineering*, *Android security policy interpretation*, and *malware behavioural analysis*. These tasks, while varied in nature, share the common requirement of extracting high-level insights from low-level data in uncertain conditions. Through our robust and flexible probabilistic analysis techniques, we are able to tackle these challenges head-on. In the following sections, we will explore each of these application areas in greater detail, demonstrating how our probabilistic analysis can be applied to address these complex tasks.

6.1 Disassembly

Disassembly, the task of reconstructing assembly code from binary code, is a fundamental process in various domains such as reverse engineering, vulnerability detection, and malware analysis. Conventionally, there are two prevalent disassembly techniques. The first, referred to as linear sweep disassemblers, operates by disassembling instructions in a sequential address order. On the other hand, traversal disassemblers follow the control flow edges, such as jumps and calls, to disassemble instructions. Both methods, while effective in their ways, present significant limitations.

Linear sweep disassemblers often grapple with false positives and negatives due to the interleavings of code and data. Conversely, traversal disassemblers struggle with indirect

control flow resulting from elements such as function pointers, virtual tables, and switch-case statements, which makes the recognition of control transfer targets exceedingly difficult. These limitations persist even in contemporary disassemblers such as BAP [17], IDA-Pro [15], OllyDbg [218], Jakstab [219], SecondWrite [220], and Dyninst [221]. Some of these can overlook up to 30% of the code in complex binaries [222].

Efforts have been made to address these issues. For instance, machine learning-based methods [101] have been proposed to recognize function entries by identifying patterns in the instruction set. Yet, they remain prone to false positives and negatives as many library functions do not adhere to specific patterns. Similarly, superset disassembly [38] was proposed as a method that disassembles at each address to generate a superset of instructions, thereby ensuring no false negatives. However, this approach causes a significant size overhead and runtime overhead on the rewritten binaries.

In light of these challenges, we emphasize the crucial role of handling uncertainty in binary analysis, particularly due to the lack of symbolic information. Our core proposition involves employing probabilistic models to handle such uncertainty and subsequently, carry out probabilistic inference to guide the disassembly of subject binaries. We present a disassembly algorithm that computes a posterior probability for each address in the code section, denoting the likelihood of that address indicating a true positive instruction (i.e., an instruction generated by the compiler).

In our approach, we disassemble the binary at each address, akin to superset disassembly, generating what we term as superset or valid instructions. We then identify relationships among these superset instructions, such as one being the transfer target of another or one defining a register that is later accessed by another. These relationships, or hints, represent semantic features that true code bodies would likely exhibit. However, they are uncertain as instructions decoded from random bytes may coincidentally possess such features. Therefore, we compute the prior probabilities of these hints using apriori probability analysis.

We propose an algorithm that aggregates these hints and computes the posterior probabilities. The resulting disassembler offers probabilistic assurances of no false negatives, lowering the likelihood of missing a true positive instruction to less than $\frac{1}{1000}$. We have

developed two variants of our probabilistic disassembly algorithms specifically for x86 and ARM binaries.

Our x86 disassembler, evaluated on 2,064 binaries, demonstrated no false negatives and a false positive rate of 3.7%. It managed to not miss any instructions even when function entries were unavailable, with a 6.8% FP. Furthermore, our evaluation on SPEC Windows PE binaries showed a noticeable improvement over traditional methods, with our tool missing no instructions compared to the 3,095 instructions missed by objdump. In the context of binary rewriting, our method outperformed the state-of-the-art superset rewriting technique, reducing the size of the rewritten binary by about 47% and improving the runtime speed of the rewritten binary by 3%.

On the other hand, our ARM disassembler was evaluated on over 5,000 binaries, covering a wide range of compilation conditions, including different options, architecture, instruction set configurations, online sources, and even obfuscation. When compared with five well-known tools (Ghidra [39], IDA [15], P-Disasm [60], XDA [118], and D-Disasm [64]), our tool consistently outperformed them, consistently achieving an F1 score over 95% in the majority of the cases. The resilience of our tool was evident even when faced with significant obfuscation; while other tools scored F1 scores in the range of 2.83-56.45%, our tool managed an impressive 78.16-88.72%.

Furthermore, our case study demonstrated the potential benefits of using our disassembler in downstream binary rewriting tasks. Our tool resulted in fewer execution failures and more accurate coverage reports, contributing to overall improved efficiency.

The technical details and inner workings of our disassembler are thoroughly discussed in our papers [60, 223]. Our work underscores the necessity of a probabilistic approach when dealing with uncertainty in disassembly, demonstrating considerable improvements in both efficiency and accuracy.

6.2 Network Protocol Reverse Engineering

The undertaking of reverse engineering network protocols poses a significant challenge within the realm of cybersecurity. It has been observed that several applications, central

to the interests of security analysts, typically employ their unique, undocumented communication protocols. For instance, autonomous vehicles adopt protocols like CAN bus and FlexRay, control systems incorporate Modbus and DNP3, while online chatting and conferencing applications engage their customized protocols. Various security analyses, such as static/symbolic vulnerability scanning [224, 225], exploit generation [226, 227], fuzzing [96, 228–230], attack detection [231, 232], and malware behavior analysis [233, 234] necessitate a detailed modeling of the network protocol. For instance, in the context of fuzzing, seeding input generation relies critically on understanding the protocol of a networking application. Malware analysis often entails crafting well-structured messages to the Command and Control (C&C) server to invoke concealed behaviors [235, 236]. Furthermore, static/symbolic analysis requires accurate modeling of networking functions to avoid the production of excessive false positives.

Existing techniques for protocol reverse engineering can be organized into a few categories. One approach capitalizes on program analysis [13, 14, 33, 237–239], where the rich semantics of the application’s implementation are dissected to achieve high accuracy. However, this method typically requires access to program binaries, often an impractical requirement due to factors such as the protective mechanisms over IoT firmware, the difficulties associated with dynamic analysis of obfuscated or packed binaries, or the elusive nature of server-side binaries for a client application. Consequently, an alternative category centers on utilizing network traces, often obtainable by network eavesdropping. This category includes two main methodologies for network trace-based reverse engineering: alignment-based (such as PIP [240], ScritGen [241], and Netzob [242]) and token-based techniques (such as Veritas [243] and Discoverer [244]). Alignment-based methods utilize sequence alignment algorithms to align message pairs and calculate similarity scores. Message clusters are formed based on these scores, with formats derived from analyzing the commonality within the clusters. However, the wide variance in message contents may significantly compromise the alignment quality, posing issues for further analysis. Token-based methods propose initial message tokenization (e.g., into textual fields and binary fields) before alignment to reduce variations. Furthermore, these techniques often require token identifiers, which may be non-existent for binary protocols, or may generate an excessive number of clusters due to

deterministic heuristic reliance. In other words, tokenization based on ad-hoc rules may not be applicable in numerous cases, leading existing techniques to often yield incorrect results due to the failure to model such uncertainties.

We observe that pinpointing the *keyword* field, which inherently determines a message’s type, is a fundamental aspect of network protocol reverse engineering. Various heuristics are available to assist in locating such keywords, but they inherently come with a degree of uncertainty. Moreover, the reverse engineering processes for both the client side and the server side can be executed simultaneously to harness their robust correspondences and create a synergistic effect. Based on these observations, we introduce a novel probabilistic methodology for reverse engineering network protocols. This approach is entirely rooted in network trace data, thus eliminating the necessity for access to source or binary code. More specifically, our technique employs *multiple sequence alignment* (MSA) [245], a strategy frequently used in bioinformatics, to circumvent the costly pairwise alignment in conventional methods. We perform a conservative alignment on all the messages initially. This comprehensive initial step reveals the common structure shared by all messages, including the keyword field, as it needs to be parsed before any type-specific parsing can take place. Once we complete the alignment, we use a probabilistic method to identify the potential keyword among the aligned fields. Acknowledging the inherent uncertainty, we introduce a random boolean variable to speculate if a field is indeed the keyword. All messages are then tentatively classified based on the values of this potential keyword. We can gain insights from the resulting clusters, such as the degree of similarity among messages within a cluster, and whether corresponding messages from the client side and the server side group into corresponding clusters. We introduce additional random variables to represent our confidence in these observations. By considering the correlations between the keyword variable and observation variables, we establish a joint probability distribution. We can then compute the posterior marginal probabilities for the keyword variables, which gives us an indication of the likelihood of each field being the true keyword. Once we identify the keyword, we can group messages based on their keyword values. The subsequent alignment and analysis of messages within these clusters will reveal type-specific structures.

To assess the efficacy of our novel approach, we put it through rigorous testing across ten protocols commonly adopted in competing projects. The results indicate that our tool attains an impressive 100% homogeneity and 97.9% completeness. This is considerably higher than the performances of leading existing techniques, which only yield roughly 92% homogeneity and 52.3% completeness. To further illustrate its wide applicability, we employed our tool to reverse engineer wireless physical-layer protocols, as well as a variety of unidentified protocols utilized in real-world IoT devices. We further substantiated the practical utility of our approach through two case studies: (i) we successfully reverse engineered the protocol for Google Nest, a popular IoT smart app. This enabled us to exert control over an A/C unit managed by the app, showcasing the real-world effectiveness of our technique; and (ii) we reverse engineered the command and control (C&C) protocol of a recent strain of malware, unveiling its hitherto hidden malevolent behaviors.

For a comprehensive understanding of the technical details and functioning of our tool, we direct the reader to our paper [246].

6.3 Android Security Policy Interpretation

Access control systems are often susceptible to security policy anomalies, including inconsistent enforcement of security policies. The Android security model is a case in point. Several research endeavors [247–249] have exposed access control inconsistencies within the Android framework, which houses framework system services and implements Application Programming Interfaces (APIs). Such inconsistencies arise when a sensitive resource (e.g., a field access, internal method, or API invocation) demands more stringent access control enforcement along one path than another. This disparity creates opportunities for malicious third-party application developers to exploit the less-protected pathway to access sensitive resources.

Previous research offers valuable, albeit approximate, solutions for detecting access control inconsistencies at the framework level. Tools like Kratos [247], AceDroid [248], and ACMiner [249] employ *convergence analysis* to evaluate the sufficiency of access control enforcement. These tools follow various paths to a *reachable shared convergence point*, where

they extract and compare the implemented access control measures to pinpoint inconsistencies. AceDroid refines the approach of Kratos by modeling and normalizing access control checks, thereby reducing the likelihood of false alarms. Recent approaches, such as FReD [250] and IAceFinder [251], strive to uncover access control inconsistencies at the framework level by exploiting security specifications across different layers of the Android software stack. FReD identifies conflicting access control requirements for APIs by comparing them against the permissions of their reachable Linux-layer files. In contrast, IAceFinder compares the enforcement of access control in both Java and native contexts to detect discrepancies between these contexts.

While existing research in this domain has provided valuable insights, it is important to note that current works do possess notable limitations. First, cross-layer inconsistency detection approaches are somewhat restrictive as they can only uncover vulnerabilities in APIs with specific implementations, such as APIs that access files as in FReD [250], or APIs that reach a JNI interface as with IAceFinder [251]. Second, even though in-framework inconsistency detection methods offer a larger landscape for examining access control due to the significant number of reachable resources at the framework layer, their underlying detection methodology is fundamentally simplistic. This simplicity often results in inaccurate findings unless further heuristics are implemented. Specifically, these tools rely on the assumption that two APIs converging on an instruction (such as field update or method invocation) are related and therefore require analogous protections. However, the convergence point might merely be auxiliary to the main functionality and hence, is likely *irrelevant* to the enforced access control. Overlooking the relevance of the convergence point can lead to a high rate of false positives. Furthermore, these tools depend exclusively on a *reachability* analysis to connect resources and deduce their access control. Yet, we have observed that Android resources are also interconnected through implicit structural, semantic, and data-flow relationships. For instance, a data-flow between two resources may suggest they require similar protections. Likewise, a naming similarity between a protected API and a reachable resource could indicate that the resource likely requires the API's protection. Modeling these implicit relationships can help expose new inconsistencies.

A fundamental limitation with existing tools is that *statically determining the necessary protections for specific resources is highly imprecise*. In a given Android API, a protection check may precede both security-relevant and non-security-relevant resources, leading to ambiguity. Simultaneously, inferring an access control implication from an implicit relationship linking resources, such as a naming similarity, inherently involves a certain level of uncertainty.

In this research, we reframe the inconsistency detection problem by incorporating probabilistic inference to account for inherent uncertainties. Rather than asserting precise correlations between resources and access control (such as, resource r necessitates protection p), we propose probabilistic associations (namely, resource r may need protection p with a confidence level of c). Our approach operates as follows: initially, we carry out a static analysis of each Android API to accumulate *basic access control facts* via a path-sensitive analysis. These facts correlate a resource r within the API to a protection p , constituting a collection of co-joined security constraints derived from discerned control dependencies. Each unique correlation is subsequently assigned a prior probability value, which signifies our degree of belief in the access control implication. We then propagate these initially assigned protections to other resources via *implication constraints*. These constraints encode the *statically observed* structural, semantic, and data-flow relationships that connect resources and facilitate the propagation of their protections. To accommodate the intrinsic uncertainty, this propagation is carried out probabilistically. Lastly, the probabilistic inference engine amalgamates the statically collected basic facts, observations, and constraints to provide a high-confidence protection recommendation for a resource. Depending on the type and quantity of facts and observations, the inference refines the initial probabilities and mitigates uncertainties. The resulting probabilistic protection recommendations can then be utilized naturally to detect access control inconsistencies.

We have implemented our proposed static analysis and probabilistic inference approach within an analysis pipeline. The evaluation of our tool highlights its efficiency in formulating protection recommendations for resources that demonstrate sufficient facts and observations. Our tool has shown remarkable performance, predicting *normalized* protections that are analogous to those implemented by AOSP with an accuracy rate reaching 84%. Moreover, our

evaluation indicates that our methodology is proficient at identifying inconsistencies. We applied our tool to scrutinize three custom images from Amazon, Xiaomi, and LG, identifying 26 veritable inconsistencies. Notably, 10 of these inconsistencies were uniquely uncovered by our tool. To further demonstrate the security implications of these inconsistencies, we constructed end-to-end Proof of Concepts (PoCs) for 8 of them. One standout observation is that a single instance of an implicit relationship led to the exposure of 118 APIs, resulting in substantial security risks, such as acquiring permissions at runtime, enforcing a recovery password, and others. We responsibly disclosed these vulnerabilities to the concerned vendors. Each vulnerability has been acknowledged and subsequently rectified.

For a comprehensive understanding and technical specifics of our tool, please refer to our paper [252].

6.4 Malware Behavioural Analysis

The rapid evolution and sophistication of malware represent a significant and ever-evolving security risk. Recent attacks highlight a shift in strategy from malware authors, who now aim for stealth and leave lighter footprints. For instance, fileless malware [253, 254] leverages in-built tools and features to infiltrate a host without requiring the installation of malicious software. Clickless infections [255] bypass end-user interaction by exploiting shared access points and remote execution vulnerabilities. Moreover, cryptocurrency malware [256, 257] enables attackers to generate substantial revenues illicitly by executing mining algorithms using the system resources of victims. According to [258], a vast cryptocurrency mining botnet accrued an astounding \$3 million revenue in 2018. This evolving threat landscape implies that malicious payloads now present themselves quite differently compared to their traditional counterparts. Consequently, a pressing challenge faced by the security community today involves comprehending and analyzing emerging malware behavior to avert potentially epidemic consequences.

One widely employed approach to comprehend malware behavior is its execution within a sandbox. However, this presents a significant challenge as the necessary environment or setup might not be available (for instance, if the Command and Control server is down or

critical libraries are missing), thereby hindering the execution of the malware. Furthermore, modern malware often employs various cloaking techniques, such as packing, time-bomb, logic bomb, and VM/debugger detectors. These techniques set very specific temporal and contextual conditions for payload release and prevent execution when the malware is under surveillance.

In light of these challenges, researchers proposed a technique called "forced-execution" (X-Force) [32]. X-Force bypasses these malware self-protection mechanisms by force-setting the outcomes of specific conditional instructions, such as those checking triggering conditions. Considering that forced execution paths could lead to corrupted states and subsequent exceptions, X-Force incorporates a "crash-free execution model" that allocates a new memory block on-demand whenever there is an invalid pointer dereference. However, X-Force has its shortcomings: it's a resource-intensive technique that is challenging to deploy in practical scenarios. Specifically, to maintain program semantics, when X-Force rectifies an invalid pointer variable (by assigning a newly allocated memory block to the variable), it needs to update all correlated pointer variables. This requirement mandates the tracking of all memory operations (to detect invalid accesses) and all move/addition/subtraction operations (to keep tabs on pointer variable correlations/aliases). Such tracking incurs significant overhead and is hard to implement correctly due to the complexity of the instruction set and the numerous corner situations to consider (e.g., in computing pointer relations). Consequently, the original X-Force does not support tracing into library functions.

Inspired by the potential benefits of probabilistic analysis, we propose a practical forced execution technique that bypasses the need for tracking individual memory or arithmetic instructions and obviates the need for on-demand memory allocation. This forced execution is very close to a native execution, naturally handling libraries and dynamically generated code. Specifically, it accomplishes crash-free execution (with probabilistic guarantees) via an innovative memory pre-planning phase. This phase pre-allocates a memory region starting from address 0, filling it with carefully designed random values. These values are crafted in such a way that if they are interpreted as addresses and subsequently dereferenced, they fall within the pre-allocated region and don't trigger an exception. Moreover, they present diverse random values such that semantically unrelated pointer variables are unlikely to

dereference the same random address, thereby preventing bogus program dependencies and corrupted states. An execution engine is developed to systematically explore different paths by force-setting varying sets of branch outcomes. For each path, multiple processes are launched to execute the path with diverse randomized memory pre-planning schemes, further decreasing the likelihood of coincidental failures. The results from these processes are consolidated to produce the results for the particular path. The engine then proceeds to the next path.

Our technique has been implemented into a prototype and has undergone rigorous testing on SPEC2000 programs, which include software like `gcc`, and a diverse array of 200 contemporary real-world malware samples. The results have been overwhelmingly positive, demonstrating our tool as a highly efficient and potent forced execution technique. When contrasted with X-Force, our approach outperforms it significantly. Specifically, our technique is faster by a factor of 84, and the rates of false positives (FP) and false negatives (FN) concerning dependence analysis are reduced by 650% and 10% respectively. Furthermore, our approach allows the detection of 102% more malicious behaviours during malware analysis. Moreover, it also significantly outperforms recent commercial and academic malware analysis engines like Cuckoo [23] and Padawan [259], offering more comprehensive and accurate analysis capabilities.

These results present strong evidence of the practical benefits of our proposed technique, showing that probabilistic analysis can be a powerful tool for uncovering and understanding malware behaviour. For a more in-depth discussion of the methodology and findings, please refer to our paper [58].

7. RELATED WORK

7.1 Program Analysis.

Program Dependence and Point-to Analysis. Program dependence analysis [260–269] are widely studied. Most of these techniques require source code. In addition, many existing works also consider control dependence whereas BDA only focuses on data dependence. Our technique is also related to points-to analysis [270–278] that addresses a similar problem. The difference lies in that our analysis does not require symbol information and hence is more difficult. Some techniques aim to reduce the runtime complexity of path-sensitive analyses [279, 280]. In contrast, our technique is sampling based. We believe BDA is complementary to existing work.

Force Execution. Force-execution [32] concretely executes a binary along different paths, by force-setting branch outcomes. It features an expensive execution engine that recovers from exceptions caused by violations of path feasibility. Due to its cost, force-execution has difficulty covering long paths. To address the above limitation, [58] propose a light-weight force-execution technique with probabilistic memory pre-planning. However, their context-insensitive path exploration strategy only focuses on predicates, leading to accuracy loss in dependence analysis.

Random Interpretation. Our underpinning technique, BDA, is related to random interpretation, a well-known probabilistic program analysis technique used in precise interprocedural analysis [281], global value numbering [282] and discovering affine equalities [283]. It features a randomized abstract interpretation that executes both branches of a conditional predicate on each run and performs a randomized affine combination at join points. However, such an affine combination is limited for numerical operations and hard to scale to binary program dependence analysis. Compared with these works, our per-path interpretation is more like concrete execution with higher accuracy and scalability.

7.2 Binary Analysis.

Binary analysis could be static [43, 284, 285], dynamic [33, 44, 286] or hybrid [287, 288]. It has a wide range of applications, such as IoT firmware security [289–294], memory forensics [295, 296], malware analysis [297], and auto-exploit [298, 299]. A large body of works focus on function entry identification [70], which is the fundamental but challenging tasks of binary analysis.

Memory Dependence. Alto [24] and VSA [19] aim to provide a sound solution to identifying aliases among memory accesses. Compared to these two, our technique is sampling based and per-sample abstract interpretation based, and hence features better precision (with probabilistic guarantee under assumption) and scalability, as shown by our results. Recently, machine learning is extensively used in binary analysis, e.g., identifying function boundary [120], pinpointing function type signature [133], and detecting similar binary code [127, 300]. In particular, [301] use LSTM to distinguish the different types of memory regions in VSA analysis. However, it does not change the core of VSA.

Variable Recovery and Type Inference. Most related to OSPREY are the studies that focus on binary variable recovery and type inference [33, 42–44]. Specifically, TIE [43] and REWARD [33] perform static and dynamic analysis to recover type information, respectively. Howard [44] improves REWARDS using heuristics to resolve conflicts. Angr [42] leverages symbolic execution to recover variables.

Decompilation. Our works are also related to decompilation techniques. *Phoenix* [40] is a security-analysis-oriented decompiler. With correctness guarantees, it aims to recover abstraction as much as possible to minimize the complexity that must be handled in downstream security analysis. To achieve this, it proposes a new control-flow recovery algorithm via semantics-preserving structural analysis and iterative refinement. Since it focuses on control-flow recovery, BDA and OSPREY are complementary.

Binary-only Fuzzing. Closely related to STOCHFuzz is binary-only fuzzing that targets on closed-source software which has only binary executables available [64, 78, 88, 89, 92, 96–99]. As aforementioned in Chapter 4, these works either rely on expensive operations or make impractical assumptions, limiting their wide adoption on real-world stripped binaries.

7.3 Probabilistic Program Analysis.

Probabilistic techniques have been increasingly used in program analysis in recent years. Probabilistic symbolic execution [302, 303] quantifies how likely it is to reach certain program points. Probabilistic model checking [304–306] encodes the probability of making a transition between states and entails computation of the likelihood that a target system satisfies a given property. Probabilistic disassembling [60] computes a probability for each address in the code space, which indicates the likelihood of the address representing a true positive instruction. Probabilistic type inference uses probabilistic graph models to infer data type [67]. There are also works on using MCMC type of sampling to derive analysis information such as memory access pattern for race detection [307] and leak detection [308], and runtime events for program understanding [309, 310]. Most of them are concrete execution based. By introducing stochastic algorithms, those hard-to-solve problems using traditional program analysis techniques can be (partially) solved in a light-weight manner, whose correctness has probabilistic guarantees under practical assumptions. Our techniques also belong to probabilistic program analysis. Specifically, BDA features a novel unbiased path sampling algorithm and leverages abstract interpretation to study the memory dependence in stripped binary code, OSPREY enforces probabilistic type inference based on the BDA-collected hints about the usage of variables and data structure fields, and STOCHFuzz leverages probabilistic analysis to aggregate evidence through many sample runs and improve rewriting on-the-fly.

7.4 N-version Programming.

N-version programming [311] is a software fault-tolerance technique, in which multiple variants of a program are executed in parallel and the results of individual executions are aggregated to reduce the likelihood of errors. It has been adopted to ensure memory safety [58, 312], concurrency security [313, 314], and computing correctness [315, 316], etc. UnTracer [317] continuously modifies target programs on the fly during fuzzing using source instrumentation so that they self-report when a test case causes new coverage, in order to improve fuzzing efficiency. Inspired by these works, STOCHFuzz also uses many versions of

rewritten binaries whose validity can be approved/disapproved by numerous fuzzing runs. The difference lies that `STOCHFuzz` is driven by a rigorous probability analysis that updates probabilities on-the-fly. Our idea of disassembling at all addresses is inspired by `Superset Disassembly` [38], which however does not leverage probabilities.

8. CONCLUSION

In this dissertation, we address the inherent uncertainty in binary analysis by developing a novel probabilistic analysis methodology, founded upon program sampling and probabilistic inference principles. Additionally, we introduce an iterative refinement architecture to enhance the effectiveness of the proposed probabilistic analysis when applicable to downstream applications. By employing the proposed methodology, we demonstrate its efficacy through three prominent binary analysis tasks: binary program dependence analysis, variable and data structure recovery, and effective and efficient binary-only fuzzing. Our methodology yields promising results in each of these tasks. We further discuss the comparison between our approach and data-driven approaches, as well as the potential applicability of our methodology to problems in other domains. All proposed solutions have successfully undergone code delivery and evaluation by the Office of Naval Research (ONR).

To address the challenge of binary dependence analysis, we introduce BDA, a practical and scalable technique featuring a novel unbiased whole-program path sampling algorithm and per-path abstract interpretation. Given certain assumptions, our technique provides a probabilistic guarantee for disclosing dependence relations. Experimental results demonstrate that our technique substantially advances the state-of-the-art, such as value set analysis, and improves performance.

In order to recover variables and data structures from stripped binaries, we devise a novel probabilistic analysis technique based on BDA. This technique employs random variables to denote the likelihood of recovery results, enabling the organic integration of numerous hints while considering inherent uncertainty. We develop a customized and optimized probabilistic constraint solving technique to address these constraints. Our experiments reveal that our technique significantly outperforms the state-of-the-art and enhances two downstream analyses.

To facilitate effective and efficient binary-only fuzzing, we devise a new fuzzing technique for stripped binaries featuring a novel incremental and stochastic rewriting technique that piggybacks on the fuzzing procedure. This approach capitalizes on the multitude of trial-and-error opportunities presented by numerous fuzzing runs to incrementally improve rewriting

accuracy. Our technique offers probabilistic guarantees on soundness, and empirical results indicate that it surpasses state-of-the-art binary-only fuzzers in terms of either soundness or overhead.

In comparison to DL-based approaches, we assert that although such strategies can reach performance levels similar to our method, they inherently lack explainability and are prone to adversarial exploitation. Our research emphasizes the value of harnessing domain-specific knowledge and logical reasoning in the realm of binary analysis. We substantiate our claim by proposing a novel attack, characterized by a trigger generation technique that causes models to misclassify and a trigger injection technique that preserves input program semantics. This attack achieves over 90% success rate on state-of-the-art Deep Learning models. The outcomes of our investigation underscore the potential of our probabilistic approach in binary analysis tasks, suggesting the potential benefits of integrating DL techniques with our method.

To evaluate the applicability of our proposed methodology for addressing problems in other domains, we examine its potential for tackling challenges in areas exhibited uncertain natures and attempts to recover high-level semantics from low-level representations. We explore several problems in depth, including disassembly, network protocol reverse engineering, Android security policy analysis, and malware analysis. Our findings suggest that the probabilistic analysis approach can be widely adapted when faced with uncertainty.

REFERENCES

- [1] J. P. Loyall and S. A. Mathisen, “Using dependence analysis to support the software maintenance process,” in *Proceedings of the Conference on Software Maintenance, ICSM 1993, Montréal, Quebec, Canada, September 1993*, 1993, pp. 282–291.
- [2] K. B. Gallagher and J. R. Lyle, “Using program slicing in software maintenance,” *IEEE Trans. Software Eng.*, vol. 17, no. 8, pp. 751–761, 1991.
- [3] J. Zeng, Y. Fu, K. A. Miller, Z. Lin, X. Zhang, and D. Xu, “Obfuscation resilient binary code reuse through trace-oriented programming,” in *2013 ACM SIGSAC Conference on Computer and Communications Security, CCS’13, Berlin, Germany, November 4-8, 2013*, 2013, pp. 487–498.
- [4] A. Sæbjørnsen, J. Willcock, T. Panas, D. J. Quinlan, and Z. Su, “Detecting code clones in binary executables,” in *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis, ISSTA 2009, Chicago, IL, USA, July 19-23, 2009*, 2009, pp. 117–128.
- [5] M. Payer, A. Barresi, and T. R. Gross, “Fine-grained control-flow integrity through binary hardening,” in *Detection of Intrusions and Malware, and Vulnerability Assessment - 12th International Conference, DIMVA 2015, Milan, Italy, July 9-10, 2015, Proceedings*, 2015, pp. 144–164.
- [6] S. Wang, W. Wang, Q. Bao, P. Wang, X. Wang, and D. Wu, “Binary code retrofitting and hardening using SGX,” in *Proceedings of the 2017 Workshop on Forming an Ecosystem Around Software Transformation, FEAST@CCS 2017, Dallas, TX, USA, November 3, 2017*, 2017, pp. 43–49.
- [7] A. Quach, A. Prakash, and L. Yan, “Debloating software through piece-wise compilation and loading,” in *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018.*, 2018, pp. 869–886.
- [8] K. Ferles, V. Wüstholtz, M. Christakis, and I. Dillig, “Failure-directed program trimming,” in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017, Paderborn, Germany, September 4-8, 2017*, 2017, pp. 174–185.

- [9] Y. Li, B. Chen, M. Chandramohan, S. Lin, Y. Liu, and A. Tiu, “Steelix: Program-state based binary fuzzing,” in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017, Paderborn, Germany, September 4-8, 2017*, 2017, pp. 627–637.
- [10] S. Rawat, V. Jain, A. Kumar, L. Cojocar, C. Giuffrida, and H. Bos, “Vuzzer: Application-aware evolutionary fuzzing,” in *NDSS*, vol. 17, 2017, pp. 1–14.
- [11] D. X. Song *et al.*, “Bitblaze: A new approach to computer security via binary analysis,” in *Information Systems Security, 4th International Conference, ICISS 2008, Hyderabad, India, December 16-20, 2008. Proceedings*, 2008, pp. 1–25.
- [12] H. Yin, D. X. Song, M. Egele, C. Kruegel, and E. Kirda, “Panorama: Capturing system-wide information flow for malware detection and analysis,” in *Proceedings of the 2007 ACM Conference on Computer and Communications Security, CCS 2007, Alexandria, Virginia, USA, October 28-31, 2007*, 2007, pp. 116–127.
- [13] J. Caballero, H. Yin, Z. Liang, and D. Song, “Polyglot: Automatic extraction of protocol message format using dynamic binary analysis,” in *Proceedings of the 14th ACM Conference on Computer and Communications Security*, 2007, pp. 317–329.
- [14] Z. Lin, X. Jiang, D. Xu, and X. Zhang, “Automatic protocol format reverse engineering through context-aware monitored execution,” in *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, vol. 8, 2008, pp. 1–15.
- [15] <https://www.hex-rays.com/products/ida>.
- [16] <https://www.grammatech.com/products/codesurfer>.
- [17] D. Brumley, I. Jager, T. Avgerinos, and E. J. Schwartz, “Bap: A binary analysis platform,” in *International Conference on Computer Aided Verification*, Springer, 2011, pp. 463–469.
- [18] <https://angr.io/>.
- [19] G. Balakrishnan and T. Reps, “Analyzing memory accesses in x86 executables,” in *International conference on compiler construction*, Springer, 2004, pp. 5–23.
- [20] <http://www.spec2000.com/>.

- [21] E. Cozzi, M. Graziano, Y. Fratantonio, and D. Balzarotti, “Understanding linux malware,” in *2018 IEEE Symposium on Security and Privacy, SP 2018, Proceedings, 21-23 May 2018, San Francisco, California, USA*, 2018, pp. 161–175.
- [22] Z. Zhang, W. You, G. Tao, G. Wei, Y. Kwon, and X. Zhang, “Bda: Practical dependence analysis for binary executables by unbiased whole-program path sampling and per-path abstract interpretation,” *Proceedings of the ACM on Programming Languages*, vol. 3, no. OOPSLA, pp. 1–31, 2019.
- [23] <https://cuckoosandbox.org/>.
- [24] S. K. Debray, R. Muth, and M. Weippert, “Alias analysis of executable code,” in *POPL ’98, Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Diego, CA, USA, January 19-21, 1998*, 1998, pp. 12–24.
- [25] J. Yang and R. Gupta, “Frequent value locality and its applications,” *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 1, no. 1, pp. 79–105, 2002.
- [26] T. Ball and J. R. Larus, “Efficient path profiling,” in *Proceedings of the 29th annual ACM/IEEE international symposium on Microarchitecture*, IEEE Computer Society, 1996, pp. 46–57.
- [27] J. Lumbroso, “Optimal discrete uniform generation from coin flips, and applications,” *arXiv preprint arXiv:1304.1916*, 2013.
- [28] O. Lhoták and K. A. Chung, “Points-to analysis with efficient strong updates,” in *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011*, 2011, pp. 3–16.
- [29] <https://rada.re/r/>.
- [30] <https://www.virustotal.com/>.
- [31] R. Muth, S. Debray, S. Watterson, K. De Bosschere, and V. E. E. Informatiesystemen, “Alto: A link-time optimizer for the dec alpha,” 1998.

- [32] F. Peng, Z. Deng, X. Zhang, D. Xu, Z. Lin, and Z. Su, “X-force: Force-executing binary programs for security applications,” in *Proceedings of the 23rd USENIX Security Symposium, San Diego, CA, USA, August 20-22, 2014.*, 2014, pp. 829–844.
- [33] Z. Lin, X. Zhang, and D. Xu, “Automatic reverse engineering of data structures from binary execution,” in *Proceedings of the 11th Annual Information Security Symposium*, 2010, pp. 1–1.
- [34] N. Carlini, A. Barresi, M. Payer, D. Wagner, and T. R. Gross, “Control-flow bending: On the effectiveness of control-flow integrity,” in *24th {USENIX} Security Symposium ({USENIX} Security 15)*, 2015, pp. 161–176.
- [35] J.-P. Martin, M. Hicks, M. Costa, P. Akritidis, and M. Castro, “Dynamically checking ownership policies in concurrent c/c++ programs,” *ACM Sigplan Notices*, vol. 45, no. 1, pp. 457–470, 2010.
- [36] M. Carbone, W. Cui, L. Lu, W. Lee, M. Peinado, and X. Jiang, “Mapping kernel objects to enable systematic integrity checking,” in *Proceedings of the 16th ACM conference on Computer and communications security*, 2009, pp. 555–565.
- [37] K. Fawaz, H. Feng, and K. G. Shin, “Anatomization and protection of mobile apps location privacy threats,” in *24th {USENIX} Security Symposium ({USENIX} Security 15)*, 2015, pp. 753–768.
- [38] E. Bauman, Z. Lin, K. W. Hamlen, *et al.*, “Superset disassembly: Statically rewriting x86 binaries without heuristics,” in *NDSS*, 2018.
- [39] <https://ghidra-sre.org/>.
- [40] D. Brumley, J. Lee, E. J. Schwartz, and M. Woo, “Native x86 decompilation using semantics-preserving structural analysis and iterative control-flow structuring,” in *Presented as part of the 22nd USENIX Security Symposium (USENIX Security 13)*, 2013, pp. 353–368.
- [41] C. Qian, H. Hu, M. Alharthi, P. H. Chung, T. Kim, and W. Lee, “{Razor}: A framework for post-deployment software debloating,” in *28th {USENIX} Security Symposium ({USENIX} Security 19)*, 2019, pp. 1733–1750.
- [42] Y. Shoshitaishvili *et al.*, “Sok:(state of) the art of war: Offensive techniques in binary analysis,” in *2016 IEEE Symposium on Security and Privacy (SP)*, IEEE, 2016, pp. 138–157.

- [43] J. Lee, T. Avgerinos, and D. Brumley, “TIE: principled reverse engineering of types in binary programs,” in *Proceedings of the Network and Distributed System Security Symposium, NDSS 2011, San Diego, California, USA, 6th February - 9th February 2011*, 2011.
- [44] A. Slowinska, T. Stancescu, and H. Bos, “Howard: A dynamic excavator for reverse engineering data structures,” in *Proceedings of the Network and Distributed System Security Symposium, NDSS 2011, San Diego, California, USA, 6th February - 9th February 2011*, 2011.
- [45] Z. Zhang *et al.*, “Osprey: Recovery of variable and data structure via probabilistic analysis for stripped binary,” in *2021 IEEE Symposium on Security and Privacy (SP)*, IEEE, 2021, pp. 813–832.
- [46] G. Balakrishnan and T. Reps, “Wysinwyx: What you see is not what you execute,” *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 32, no. 6, pp. 1–84, 2010.
- [47] https://github.com/NationalSecurityAgency/ghidra/blob/master/Ghidra/Features/Decompiler/ghidra_scripts/CreateStructure.java#L25.
- [48] E. De Cristofaro, J.-M. Bohli, and D. Westhoff, “Fair: Fuzzy-based aggregation providing in-network resilience for real-time wireless sensor networks,” in *Proceedings of the second ACM conference on Wireless network security*, 2009, pp. 253–260.
- [49] H. Peng, Y. Shoshitaishvili, and M. Payer, “T-fuzz: Fuzzing by program transformation,” in *2018 IEEE Symposium on Security and Privacy (SP)*, IEEE, 2018, pp. 697–710.
- [50] <http://lcamtuf.coredump.cx/afl>.
- [51] D. She, K. Pei, D. Epstein, J. Yang, B. Ray, and S. Jana, “NEUZZ: efficient fuzzing with neural program smoothing,” in *2019 IEEE Symposium on Security and Privacy, SP 2019, San Francisco, CA, USA, May 19-23, 2019*, 2019, pp. 803–817.
- [52] C. Cadar, D. Dunbar, and D. R. Engler, “KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs,” in *8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008, December 8-10, 2008, San Diego, California, USA, Proceedings*, 2008, pp. 209–224.

- [53] X. Ge, K. Taneja, T. Xie, and N. Tillmann, “Dyta: Dynamic symbolic execution guided with static verification results,” in *Proceedings of the 33rd International Conference on Software Engineering, ICSE 2011, Waikiki, Honolulu , HI, USA, May 21-28, 2011*, R. N. Taylor, H. C. Gall, and N. Medvidovic, Eds., ACM, 2011, pp. 992–994.
- [54] J. Chen, W. Hu, L. Zhang, D. Hao, S. Khurshid, and L. Zhang, “Learning to accelerate symbolic execution via code transformation,” in *32nd European Conference on Object-Oriented Programming, ECOOP 2018, July 16-21, 2018, Amsterdam, The Netherlands*, T. D. Millstein, Ed., ser. LIPIcs, vol. 109, Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2018, 6:1–6:27.
- [55] S. Person, G. Yang, N. Rungta, and S. Khurshid, “Directed incremental symbolic execution,” in *Acm Sigplan Notices*, ACM, vol. 46, 2011, pp. 504–515.
- [56] V. Chipounov, V. Kuznetsov, and G. Candea, “S2e: A platform for in-vivo multi-path analysis of software systems,” *Acm Sigplan Notices*, vol. 46, no. 3, pp. 265–278, 2011.
- [57] T. Xie, N. Tillmann, J. De Halleux, and W. Schulte, “Fitness-guided path exploration in dynamic symbolic execution,” in *2009 IEEE/IFIP International Conference on Dependable Systems & Networks*, IEEE, 2009, pp. 359–368.
- [58] W. You *et al.*, “Pmp: Cost-effective forced execution with probabilistic memory pre-planning,” in *2020 IEEE Symposium on Security and Privacy (SP)*, 2020, pp. 381–398.
- [59] B. Livshits, A. V. Nori, S. K. Rajamani, and A. Banerjee, “Merlin: Specification inference for explicit information flow problems,” *ACM Sigplan Notices*, vol. 44, no. 6, pp. 75–86, 2009.
- [60] K. Miller, Y. Kwon, Y. Sun, Z. Zhang, X. Zhang, and Z. Lin, “Probabilistic disassembly,” in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, IEEE, 2019, pp. 1187–1198.
- [61] M. A. B. Khadra, D. Stoffel, and W. Kunz, “Speculative disassembly of binary code,” in *2016 International Conference on Compilers, Architectures, and Synthesis of Embedded Systems (CASES)*, IEEE, 2016, pp. 1–10.
- [62] C. Kruegel, W. Robertson, F. Valeur, and G. Vigna, “Static disassembly of obfuscated binaries,” in *USENIX security Symposium*, vol. 13, 2004, pp. 18–18.

- [63] M. Popa, “Binary code disassembly for reverse engineering,” *Journal of Mobile, Embedded and Distributed Systems*, vol. 4, no. 4, pp. 233–248, 2012.
- [64] A. Flores-Montoya and E. Schulte, “Datalog disassembly,” in *29th {USENIX} Security Symposium ({USENIX} Security 20)*, 2020.
- [65] H.-A. Loeliger, J. Dauwels, J. Hu, S. Korl, L. Ping, and F. R. Kschischang, “The factor graph approach to model-based signal processing,” *Proceedings of the IEEE*, vol. 95, no. 6, pp. 1295–1322, 2007.
- [66] N. E. Beckman and A. V. Nori, “Probabilistic, modular and scalable inference of typestate specifications,” in *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, 2011, pp. 211–221.
- [67] Z. Xu, X. Zhang, L. Chen, K. Pei, and B. Xu, “Python probabilistic type inference with natural language support,” in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2016, pp. 607–618.
- [68] T. Kremenek, P. Twohey, G. Back, A. Ng, and D. Engler, “From uncertainty to belief: Inferring the specification within,” in *Proceedings of the 7th symposium on Operating systems design and implementation*, 2006, pp. 161–176.
- [69] <https://www.gnu.org/software/coreutils/>.
- [70] D. Andriesse, X. Chen, V. Van Der Veen, A. Slowinska, and H. Bos, “An in-depth analysis of disassembly on full-scale x86/x64 binaries,” in *25th {USENIX} Security Symposium ({USENIX} Security 16)*, 2016, pp. 583–600.
- [71] https://en.wikipedia.org/wiki/F1_score.
- [72] A. V. Aho and J. D. Ullman, *Principles of compiler design*. Addison-Wesley, 1977.
- [73] P. Zhao and J. N. Amaral, “Function outlining and partial inlining,” in *17th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD’05)*, IEEE, 2005, pp. 101–108.
- [74] [https://en.wikipedia.org/wiki/Fortune_\(Unix\)](https://en.wikipedia.org/wiki/Fortune_(Unix)).

- [75] C. Lattner and V. Adve, “LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation,” in *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO’04)*, Palo Alto, California, Mar. 2004.
- [76] <https://software.intel.com/content/www/us/en/develop/articles/pin-a-dynamic-binary-instrumentation-tool.html>.
- [77] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, “Addresssanitizer: A fast address sanity checker,” in *Presented as part of the 2012 {USENIX} Annual Technical Conference ({USENIX}{ATC} 12)*, 2012, pp. 309–318.
- [78] S. Dinesh, N. Burow, D. Xu, and M. Payer, “Retrowrite: Statically instrumenting cots binaries for fuzzing and sanitization,” in *SP*, 2020.
- [79] <https://nvd.nist.gov/vuln/detail/CVE-2019-12802>.
- [80] <https://github.com/ZhangZhuoSJTU/StochFuzz>.
- [81] M. Böhme, V.-T. Pham, and A. Roychoudhury, “Coverage-based greybox fuzzing as markov chain,” in *CCS*, 2016, pp. 1032–1043.
- [82] W. You, X. Liu, S. Ma, D. Perry, X. Zhang, and B. Liang, “SLF: Fuzzing without valid seed inputs,” in *ICSE*, 2019, pp. 712–723.
- [83] <https://software.intel.com/content/www/us/en/develop/blogs/processor-tracing.html>.
- [84] <https://www.qemu.org/>.
- [85] <https://software.intel.com/content/www/us/en/develop/articles/pin-a-dynamic-binary-instrumentation-tool.html>.
- [86] R. Wartell, Y. Zhou, K. W. Hamlen, M. Kantarcioglu, and B. Thuraisingham, “Differentiating code from data in x86 binaries,” in *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*, Springer, 2011, pp. 522–536.
- [87] C. Pang *et al.*, “Sok: All you ever wanted to know about x86/x64 binary disassembly but were afraid to ask,” *arXiv preprint arXiv:2007.14266*, 2020.
- [88] <https://github.com/talos-vulndev/afl-dyninst>.

- [89] G. J. Duck, X. Gao, and A. Roychoudhury, “Binary rewriting without control flow recovery,” in *PLDI*, 2020, pp. 151–163.
- [90] Z. Zhang, W. You, G. Tao, Y. Aafer, X. Liu, and X. Zhang, “Stochfuzz: Sound and cost-effective fuzzing of stripped binaries by incremental and stochastic rewriting,” in *2021 IEEE Symposium on Security and Privacy (SP)*, IEEE, 2021, pp. 659–676.
- [91] <https://github.com/google/fuzzer-test-suite>.
- [92] G. Zhang, X. Zhou, Y. Luo, X. Wu, and E. Min, “Ptfuzz: Guided fuzzing with processor trace feedback,” *IEEE Access*, 2018.
- [93] https://github.com/google/AFL/tree/master/qemu_mode.
- [94] https://github.com/google/AFL/tree/master/llvm_mode.
- [95] C. Aschermann, S. Schumilo, A. Abbasi, and T. Holz, “Ijon: Exploring deep state spaces via fuzzing,” in *SP*, 2020, pp. 1597–1612.
- [96] Y. Chen *et al.*, “Ptrix: Efficient hardware-assisted fuzzing for cots binary,” in *Proceedings of the 2019 ACM Asia Conference on Computer and Communications Security*, 2019, pp. 633–645.
- [97] S. Schumilo, C. Aschermann, R. Gawlik, S. Schinzel, and T. Holz, “Kafk: Hardware-assisted feedback fuzzing for {os} kernels,” in *USENIX Security*, 2017, pp. 167–182.
- [98] <https://github.com/vanhauser-thc/afl-pin>.
- [99] <https://github.com/vanhauser-thc/afl-dynamorio>.
- [100] <https://github.com/GJDuck/e9afl>.
- [101] T. Bao, J. Burket, M. Woo, R. Turner, and D. Brumley, “BYTEWEIGHT: Learning to recognize functions in binary code,” in *USENIX Security*, 2014, pp. 845–860.
- [102] A. Zeller, “Yesterday, my program worked. today, it does not. why?” *ACM SIGSOFT Software engineering notes*, pp. 253–267, 1999.

- [103] J. S. Yedidia, W. T. Freeman, and Y. Weiss, “Generalized belief propagation,” in *Advances in neural information processing systems*, 2001, pp. 689–695.
- [104] F. R. Kschischang, B. J. Frey, and H.-A. Loeliger, “Factor graphs and the sum-product algorithm,” *IEEE Transactions on information theory*, vol. 47, no. 2, pp. 498–519, 2001.
- [105] K. Murphy, Y. Weiss, and M. I. Jordan, “Loopy belief propagation for approximate inference: An empirical study,” *arXiv preprint arXiv:1301.6725*, 2013.
- [106] C.-C. Hsu, C.-Y. Wu, H.-C. Hsiao, and S.-K. Huang, “Instrim: Lightweight instrumentation for coverage-guided fuzzing,” in *Symposium on Network and Distributed System Security (NDSS), Workshop on Binary Analysis Research*, 2018.
- [107] https://afl-1.readthedocs.io/en/latest/user_guide.html.
- [108] <https://www.capstone-engine.org/>.
- [109] <https://www.keystone-engine.org/>.
- [110] W. You *et al.*, “Profuzzer: On-the-fly input type probing for better zero-day vulnerability discovery,” in *SP*, 2019.
- [111] P. Zong, T. Lv, D. Wang, Z. Deng, R. Liang, and K. Chen, “Fuzzguard: Filtering out unreachable inputs in directed grey-box fuzzing through deep learning,” in *USENIX Security*, 2020, pp. 2255–2269.
- [112] <https://github.com/AFLplusplus/AFLplusplus/issues/24>.
- [113] <https://github.com/hunter-ht-2018/ptfuzzer>.
- [114] <https://www.clear.rice.edu/comp422/resources/cuda/html/cuda-binary-utilities/index.html>.
- [115] <https://www.nvidia.com/en-us/security/acknowledgements/>.
- [116] <https://ewww.io/>.
- [117] <https://github.com/ImageOptim/ImageOptim>.

- [118] K. Pei, J. Guan, D. Williams-King, J. Yang, and S. Jana, “Xda: Accurate, robust disassembly with transfer learning,” in *28th Annual Network and Distributed System Security Symposium, NDSS 2021, virtually, February 21-25, 2021*, The Internet Society, 2021.
- [119] S. Yu, Y. Qu, X. Hu, and H. Yin, “Deepdi: Learning a relational graph convolutional network model on instructions for fast and accurate disassembly,”
- [120] E. C. R. Shin, D. Song, and R. Moazzezi, “Recognizing functions in binaries with neural networks,” in *24th USENIX Security Symposium (USENIX Security 15)*, 2015, pp. 611–626.
- [121] K. Pei, Z. Xuan, J. Yang, S. Jana, and B. Ray, “Trex: Learning execution semantics from micro-traces for binary similarity,” *arXiv preprint arXiv:2012.08680*, 2020.
- [122] L. Massarelli, G. A. Di Luna, F. Petroni, R. Baldoni, and L. Querzoni, “Safe: Self-attentive function embeddings for binary similarity,” in *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, Springer, 2019, pp. 309–329.
- [123] L. Massarelli, G. A. Di Luna, F. Petroni, L. Querzoni, and R. Baldoni, “Investigating graph embedding neural networks with unsupervised features extraction for binary analysis,” in *Proceedings of the 2nd Workshop on Binary Analysis Research (BAR)*, 2019.
- [124] F. Zuo, X. Li, P. Young, L. Luo, Q. Zeng, and Z. Zhang, “Neural machine translation inspired binary code similarity comparison beyond function pairs,” *arXiv preprint arXiv:1808.04706*, 2018.
- [125] X. Zhang, W. Sun, J. Pang, F. Liu, and Z. Ma, “Similarity metric method for binary basic blocks of cross-instruction set architecture,” in *Proceedings 2020 Workshop on Binary Analysis Research*, Internet Society, 2020.
- [126] K. Redmond, L. Luo, and Q. Zeng, “A cross-architecture instruction embedding model for natural language processing-inspired binary code analysis,” *arXiv preprint arXiv:1812.09652*, 2018.
- [127] S. H. Ding, B. C. Fung, and P. Charland, “Asm2vec: Boosting static representation robustness for binary clone search against code obfuscation and compiler optimization,” in *2019 IEEE Symposium on Security and Privacy (SP)*, IEEE, 2019, pp. 472–489.

- [128] Z. Yu, R. Cao, Q. Tang, S. Nie, J. Huang, and S. Wu, “Order matters: Semantic-aware neural networks for binary code similarity detection,” in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 34, 2020, pp. 1145–1152.
- [129] K. Pei *et al.*, “Stateformer: Fine-grained type recovery from binaries using generative state modeling,” in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2021, pp. 690–702.
- [130] A. Maier, H. Gascon, C. Wressnegger, and K. Rieck, “Typeminer: Recovering types in binary programs using machine learning,” in *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, Springer, 2019, pp. 288–308.
- [131] J. He, P. Ivanov, P. Tsankov, V. Raychev, and M. Vechev, “Debin: Predicting debug information in stripped binaries,” in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018, pp. 1667–1680.
- [132] Q. Chen, J. Lacomis, E. J. Schwartz, C. Le Goues, G. Neubig, and B. Vasilescu, “Augmenting decompiler output with learned variable names and types,” *arXiv preprint arXiv:2108.06363*, 2021.
- [133] Z. L. Chua, S. Shen, P. Saxena, and Z. Liang, “Neural nets can learn function type signatures from binaries,” in *26th USENIX Security Symposium (USENIX Security 17)*, 2017, pp. 99–116.
- [134] H. Gao, S. Cheng, Y. Xue, and W. Zhang, “A lightweight framework for function name reassignment based on large-scale stripped binaries,” in *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2021, pp. 607–619.
- [135] F. Artuso, G. A. Di Luna, L. Massarelli, and L. Querzoni, “In nomine function: Naming functions in stripped binaries with neural networks,” *arXiv preprint arXiv:1912.07946*, 2019.
- [136] J. Lacomis *et al.*, “Dire: A neural approach to decompiled identifier naming,” in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, IEEE, 2019, pp. 628–639.

- [137] S. I. Popoola, R. Ande, B. Adebisi, G. Gui, M. Hammoudeh, and O. Jogunola, “Federated deep learning for zero-day botnet attack detection in iot edge devices,” *IEEE Internet of Things Journal*, 2021.
- [138] C. Do Xuan, M. H. Dao, and H. D. Nguyen, “Apt attack detection based on flow network analysis techniques using deep learning,” *Journal of Intelligent & Fuzzy Systems*, vol. 39, no. 3, pp. 4785–4801, 2020.
- [139] K. Yu *et al.*, “Securing critical infrastructures: Deep-learning-based threat detection in iiot,” *IEEE Communications Magazine*, vol. 59, no. 10, pp. 76–82, 2021.
- [140] N. Koroniotis, N. Moustafa, and E. Sitnikova, “Forensics and deep learning mechanisms for botnets in internet of things: A survey of challenges and solutions,” *IEEE Access*, vol. 7, pp. 61 764–61 785, 2019.
- [141] A. Alsaheel *et al.*, “ATLAS: A sequence-based learning approach for attack investigation,” in *30th USENIX Security Symposium (USENIX Security 21)*, 2021.
- [142] G. Wang, Y. Cui, J. Wang, L. Wu, and G. Hu, “A novel method for detecting advanced persistent threat attack based on belief rule base,” *Applied Sciences*, vol. 11, no. 21, p. 9899, 2021.
- [143] W. Wang *et al.*, “Hast-ids: Learning hierarchical spatial-temporal features using deep neural networks to improve intrusion detection,” *IEEE access*, vol. 6, pp. 1792–1806, 2017.
- [144] C. Yin, Y. Zhu, J. Fei, and X. He, “A deep learning approach for intrusion detection using recurrent neural networks,” *Ieee Access*, vol. 5, pp. 21 954–21 961, 2017.
- [145] N. Shone, T. N. Ngoc, V. D. Phai, and Q. Shi, “A deep learning approach to network intrusion detection,” *IEEE transactions on emerging topics in computational intelligence*, vol. 2, no. 1, pp. 41–50, 2018.
- [146] R. Vinayakumar, M. Alazab, K. Soman, P. Poornachandran, A. Al-Nemrat, and S. Venkatraman, “Deep learning approach for intelligent intrusion detection system,” *IEEE Access*, vol. 7, pp. 41 525–41 550, 2019.
- [147] S. Zhao, X. Ma, Y. Wang, J. Bailey, B. Li, and Y.-G. Jiang, “What do deep nets learn? class-wise patterns revealed in the input space,” *arXiv preprint arXiv:2101.06898*, 2021.

- [148] G. Tao *et al.*, “Model orthogonalization: Class distance hardening in neural networks for better security,” in *2022 IEEE Symposium on Security and Privacy (SP)*, IEEE, 2022.
- [149] T. Gu, K. Liu, B. Dolan-Gavitt, and S. Garg, “Badnets: Evaluating backdooring attacks on deep neural networks,” *IEEE Access*, 2019.
- [150] A. Shafahi *et al.*, “Poison frogs! targeted clean-label poisoning attacks on neural networks,” in *NeurIPS*, 2018.
- [151] A. Salem, R. Wen, M. Backes, S. Ma, and Y. Zhang, “Dynamic backdoor attacks against machine learning models,” *arXiv preprint arXiv:2003.03675*, 2020.
- [152] X. Zhang, Z. Zhang, and T. Wang, “Trojaning language models for fun and profit,” in *European S&P*, 2021.
- [153] X. Chen, A. Salem, M. Backes, S. Ma, and Y. Zhang, “Badnl: Backdoor attacks against nlp models,” *arXiv preprint arXiv:2006.01043*, 2020.
- [154] K. Kurita, P. Michel, and G. Neubig, “Weight poisoning attacks on pre-trained models,” in *ACL*, 2020.
- [155] I. J. Goodfellow, J. Shlens, and C. Szegedy, “Explaining and harnessing adversarial examples,” in *International Conference on Learning Representations (ICLR)*, 2015.
- [156] W. Brendel, J. Rauber, and M. Bethge, “Decision-based adversarial attacks: Reliable attacks against black-box machine learning models,” in *International Conference on Learning Representations (ICLR)*, 2018.
- [157] N. Carlini and D. Wagner, “Towards evaluating the robustness of neural networks,” in *2017 IEEE Symposium on Security and Privacy (SP)*, IEEE, 2017, pp. 39–57.
- [158] N. Carlini and D. Wagner, “Adversarial examples are not easily detected: Bypassing ten detection methods,” in *Proceedings of the 10th ACM Workshop on Artificial Intelligence and Security*, 2017, pp. 3–14.
- [159] S. Chen *et al.*, “Automated poisoning attacks and defenses in malware detection systems: An adversarial machine learning approach,” *Computers & Security*, vol. 73, pp. 326–344, 2018.

- [160] C. Xiao, J. Y. Zhu, B. Li, W. He, M. Liu, and D. Song, “Spatially transformed adversarial examples,” in *6th International Conference on Learning Representations, ICLR 2018*, 2018.
- [161] https://www.theregister.com/2022/03/26/machine_learning_malware/.
- [162] Y. Liu, X. Ma, J. Bailey, and F. Lu, “Reflection backdoor: A natural backdoor attack on deep neural networks,” in *ECCV*, 2020.
- [163] J. Lin, L. Xu, Y. Liu, and X. Zhang, “Composite backdoor attack for deep neural network by mixing existing benign features,” in *CCS*, 2020.
- [164] E. Bagdasaryan and V. Shmatikov, “Blind backdoors in deep learning models,” *arXiv preprint arXiv:2005.03823*, 2020.
- [165] J. Li, S. Ji, T. Du, B. Li, and T. Wang, “Textbugger: Generating adversarial text against real-world applications,” in *26th Annual Network and Distributed System Security Symposium*, 2019.
- [166] Z. Gong, W. Wang, B. Li, D. Song, and W.-S. Ku, “Adversarial texts with gradient methods,” *arXiv preprint arXiv:1801.07175*, 2018.
- [167] Z. Zhang *et al.*, “Pelican: Exploiting backdoors of naturally trained deep learning models in binary code analysis,” 2023.
- [168] P. Junod, J. Rinaldini, J. Wehrli, and J. Michielin, “Obfuscator-llvm—software protection for the masses,” in *2015 IEEE/ACM 1st International Workshop on Software Protection*, IEEE, 2015, pp. 3–9.
- [169] Y. Liu *et al.*, “Trojaning attack on neural networks,” in *NDSS*, 2018.
- [170] <https://binary.ninja/>.
- [171] V. Duddu, D. Samanta, D. V. Rao, and V. E. Balas, “Stealing neural networks via timing side channels,” *arXiv preprint arXiv:1812.11720*, 2018.
- [172] X. Hu *et al.*, “Deepsniffer: A dnn model extraction framework based on learning architectural hints,” in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2020, pp. 385–399.

- [173] W. Hua, Z. Zhang, and G. E. Suh, “Reverse engineering convolutional neural networks through side-channel information leaks,” in *2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC)*, IEEE, 2018, pp. 1–6.
- [174] Y. Xiang *et al.*, “Open dnn box by power side-channel attack,” *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 67, no. 11, pp. 2717–2721, 2020.
- [175] M. Yan, C. W. Fletcher, and J. Torrellas, “Cache telepathy: Leveraging shared resource attacks to learn DNN architectures,” in *29th USENIX Security Symposium (USENIX Security 20)*, 2020, pp. 2003–2020.
- [176] Y. Zhu, Y. Cheng, H. Zhou, and Y. Lu, “Hermes attack: Steal DNN models with lossless inference accuracy,” in *30th USENIX Security Symposium (USENIX Security 21)*, 2021.
- [177] N. Papernot, P. McDaniel, I. Goodfellow, S. Jha, Z. B. Celik, and A. Swami, “Practical black-box attacks against machine learning,” in *Proceedings of the 2017 ACM on Asia conference on computer and communications security*, 2017.
- [178] Y. Liu, X. Chen, C. Liu, and D. Song, “Delving into transferable adversarial examples and black-box attacks,” *arXiv preprint arXiv:1611.02770*, 2016.
- [179] A. Ilyas, L. Engstrom, A. Athalye, and J. Lin, “Black-box adversarial attacks with limited queries and information,” in *International Conference on Machine Learning*, PMLR, 2018.
- [180] S.-M. Moosavi-Dezfooli, A. Fawzi, O. Fawzi, and P. Frossard, “Universal adversarial perturbations,” in *CVPR*, 2017, pp. 1765–1773.
- [181] <https://github.com/jgamblin/Mirai-Source-Code>.
- [182] Y. Liu, G. Shen, G. Tao, Z. Wang, S. Ma, and X. Zhang, “Ex-ray: Distinguishing injected backdoor from natural features in neural networks by examining differential feature symmetry,” *arXiv preprint arXiv:2103.08820*, 2021.
- [183] C. Guo, A. Sablayrolles, H. Jégou, and D. Kiela, “Gradient-based adversarial attacks against text transformers,” *arXiv preprint arXiv:2104.13733*, 2021.
- [184] https://en.wikipedia.org/wiki/X86_calling_conventions.

- [185] R. Tofighi-Shirazi, I.-M. Asavoae, P. Elbaz-Vincent, and T.-H. Le, “Defeating opaque predicates statically through machine learning and binary analysis,” in *Proceedings of the 3rd ACM Workshop on Software Protection*, 2019, pp. 3–14.
- [186] J. Ming, D. Xu, L. Wang, and D. Wu, “Loop: Logic-oriented opaque predicate detection in obfuscated binary code,” in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, 2015, pp. 757–768.
- [187] R. K. R. Prakash, P. Amritha, and M. Sethumadhavan, “Opaque predicate detection by static analysis of binary executables,” in *International Symposium on Security in Computing and Communication*, Springer, 2017, pp. 250–258.
- [188] T. Rinsma, *Seeing through obfuscation: Interactive detection and removal of opaque predicates*, 2017.
- [189] D. Bruschi, L. Martignoni, and M. Monga, “Detecting self-mutating malware using control-flow graph matching,” in *International conference on detection of intrusions and malware, and vulnerability assessment*, Springer, 2006, pp. 129–143.
- [190] M. D. Preda, M. Christodorescu, S. Jha, and S. Debray, “A semantics-based approach to malware detection,” *ACM SIGPLAN Notices*, vol. 42, no. 1, pp. 377–388, 2007.
- [191] M. Dalla Preda, M. Madou, K. De Bosschere, and R. Giacobazzi, “Opaque predicates detection by abstract interpretation,” in *International Conference on Algebraic Methodology and Software Technology*, Springer, 2006, pp. 81–95.
- [192] L. De Moura and N. Bjørner, “Z3: An efficient smt solver,” in *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, Springer, 2008, pp. 337–340.
- [193] J. Turian, L.-A. Ratinov, and Y. Bengio, “Word representations: A simple and general method for semi-supervised learning,” in *Proceedings of the 48th Annual Meeting of the Association for Computational Linguistics*, Uppsala, Sweden: Association for Computational Linguistics, Jul. 2010, pp. 384–394. [Online]. Available: <https://aclanthology.org/P10-1040>.
- [194] M. E. Peters, W. Ammar, C. Bhagavatula, and R. Power, “Semi-supervised sequence tagging with bidirectional language models,” 2017.
- [195] M. Gardner *et al.*, “Allennlp: A deep semantic natural language processing platform,” *ACL 2018*, p. 1, 2018.

- [196] A. Biere, M. Heule, and H. van Maaren, *Handbook of satisfiability*. IOS press, 2009, vol. 185.
- [197] P. Fontaine and H.-J. Schurr, “Quantifier simplification by unification in smt,” in *International Symposium on Frontiers of Combining Systems*, Springer, 2021.
- [198] A. R. Bradley, Z. Manna, and H. B. Sipma, “Whats decidable about arrays?” In *International Workshop on Verification, Model Checking, and Abstract Interpretation*, Springer, 2006.
- [199] D. Trabish and N. Rinetzky, “Relocatable addressing model for symbolic execution,” in *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2020.
- [200] D. Kuts, “Towards symbolic pointers reasoning in dynamic symbolic execution,” in *2021 Ivannikov Memorial Workshop (IVMEM)*, IEEE, 2021.
- [201] X. Li, Y. Qu, and H. Yin, “Palmtree: Learning an assembly language model for instruction embedding,” in *CCS ’21: 2021 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, Republic of Korea, November 15 - 19, 2021*, Y. Kim, J. Kim, G. Vigna, and E. Shi, Eds., ACM, 2021, pp. 3236–3251. DOI: [10.1145/3460120.3484587](https://doi.org/10.1145/3460120.3484587). [Online]. Available: <https://doi.org/10.1145/3460120.3484587>.
- [202] Y. Zang *et al.*, “Word-level textual adversarial attacking as combinatorial optimization,” *arXiv preprint arXiv:1910.12196*, 2019.
- [203] X. Dong, A. T. Luu, R. Ji, and H. Liu, “Towards robustness against natural language word substitutions,” *arXiv preprint arXiv:2107.13541*, 2021.
- [204] F. Pierazzi, F. Pendlebury, J. Cortellazzi, and L. Cavallaro, “Intriguing properties of adversarial ml attacks in the problem space,” in *2020 IEEE Symposium on Security and Privacy (SP)*, IEEE, 2020, pp. 1332–1349.
- [205] K. Lucas, M. Sharif, L. Bauer, M. K. Reiter, and S. Shintre, “Malware makeover: Breaking ml-based static analysis by modifying executable bytes,” in *Proceedings of the 2021 ACM Asia Conference on Computer and Communications Security*, 2021.
- [206] L. Yang *et al.*, “Jigsaw puzzle: Selective backdoor attack to subvert malware classifiers,” *arXiv preprint arXiv:2202.05470*, 2022.

- [207] G. Severi, J. Meyer, S. Coull, and A. Oprea, “{Explanation-guided} backdoor poisoning attacks against malware classifiers,” in *30th USENIX Security Symposium (USENIX Security 21)*, 2021.
- [208] Z. Kan, H. Wang, L. Wu, Y. Guo, and D. X. Luo, “Automated deobfuscation of android native binary code,” *arXiv preprint arXiv:1907.06828*, 2019.
- [209] L. Coniglio, “Combining program synthesis and symbolic execution to deobfuscate binary code,” M.S. thesis, University of Twente, 2019.
- [210] Y. Guillot and A. Gazet, “Automatic binary deobfuscation,” *Journal in computer virology*, vol. 6, no. 3, 2010.
- [211] C. Jindal, C. Salls, H. Aghakhani, K. Long, C. Kruegel, and G. Vigna, “Neurlux: Dynamic malware analysis without feature engineering,” in *Proceedings of the 35th Annual Computer Security Applications Conference*, 2019.
- [212] S. Wu and X. Xiao, “ConvDroid: Lightweight neural network based android malware detection,” *Aust. J. Intell. Inf. Process. Syst.*, vol. 16, no. 3, 2019.
- [213] Y.-J. Tung and I. G. Harris, “A heuristic approach to detect opaque predicates that disrupt static disassembly,”
- [214] H. Koo and M. Polychronakis, “Juggling the gadgets: Binary-level code randomization using instruction displacement,” in *AsiaCCS*, 2016.
- [215] <https://www.spec.org/cpu2006/>.
- [216] <https://www.gnu.org/software/binutils/>.
- [217] <https://www.deepbits.com/>.
- [218] O. Yuschuk, “Ollydbg,” <http://www.ollydbg.de/>, 2007.
- [219] J. Kinder and H. Veith, “Jakstab: A static analysis platform for binaries,” in *International Conference on Computer Aided Verification*, Springer, 2008, pp. 423–427.
- [220] M. Smithson, K. Anand, A. Kotha, K. Elwazeer, N. Giles, and R. Barua, *Second write: Binary rewriting without relocation information*, University of Maryland, 2010.

- [221] A. R. Bernat and B. P. Miller, “Anywhere, any-time binary instrumentation,” in *Proceedings of the 10th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools*, ACM, 2011, pp. 9–16.
- [222] X. Meng and B. P. Miller, “Binary code is not easy,” in *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ACM, 2016, pp. 24–35.
- [223] Y. Ye, Z. Zhang, Q. Shi, Y. Aafer, and X. Zhang, “D-arm: Disassembling arm binaries by lightweight superset instruction interpretation and graph modeling,” in *2023 IEEE Symposium on Security and Privacy (SP)*, IEEE Computer Society, 2022, pp. 728–745.
- [224] E. Hoque, O. Chowdhury, S. Y. Chau, C. Nita-Rotaru, and N. Li, “Analyzing operational behavior of stateful protocol implementations for detecting semantic bugs,” in *2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2017, pp. 627–638.
- [225] K. Borgolte, C. Kruegel, and G. Vigna, “Delta: Automatic identification of unknown web-based infection campaigns,” in *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*, 2013, pp. 109–120.
- [226] M. von Hippel, C. Vick, S. Tripakis, and C. Nita-Rotaru, “Automated attacker synthesis for distributed protocols,” *arXiv preprint arXiv:2004.01220*, 2020.
- [227] T. Bao, R. Wang, Y. Shoshitaishvili, and D. Brumley, “Your exploit is mine: Automatic shellcode transplant for remote exploits,” in *2017 IEEE Symposium on Security and Privacy (SP)*, 2017, pp. 824–839.
- [228] S. Österlund, K. Razavi, H. Bos, and C. Giuffrida, “Parmesan: Sanitizer-guided greybox fuzzing,” in *29th USENIX Security Symposium (USENIX Security)*, 2020, pp. 2289–2306.
- [229] V. Jain, S. Rawat, C. Giuffrida, and H. Bos, “Tiff: Using input type inference to improve fuzzing,” in *Proceedings of the 34th Annual Computer Security Applications Conference*, 2018, pp. 505–517.
- [230] S. Jero, M. L. Pacheco, D. Goldwasser, and C. Nita-Rotaru, “Leveraging textual specifications for grammar-based fuzzing of network protocols,” in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 33, 2019, pp. 9478–9483.

- [231] A. Abdou, D. Barrera, and P. C. Van Oorschot, “What lies beneath? analyzing automated ssh bruteforce attacks,” in *International Conference on PASSWORDS*, 2015, pp. 72–91.
- [232] Y. Cao, Y. Shoshitaishvili, K. Borgolte, C. Kruegel, G. Vigna, and Y. Chen, “Protecting web-based single sign-on protocols against relying party impersonation attacks through a dedicated bi-directional authenticated secure channel,” in *International Workshop on Recent Advances in Intrusion Detection*, 2014, pp. 276–298.
- [233] G. Starnberger, C. Kruegel, and E. Kirda, “Overbot: A botnet protocol based on kademia,” in *Proceedings of the 4th International Conference on Security and Privacy in Communication Networks*, 2008, pp. 1–9.
- [234] M. Antonakakis *et al.*, “Understanding the mirai botnet,” in *26th USENIX Security Symposium (USENIX Security)*, 2017, pp. 1093–1110.
- [235] L. Bilge, D. Balzarotti, W. Robertson, E. Kirda, and C. Kruegel, “Disclosure: Detecting botnet command and control servers through large-scale netflow analysis,” in *Proceedings of the 28th Annual Computer Security Applications Conference*, 2012, pp. 129–138.
- [236] P. Wurzinger, L. Bilge, T. Holz, J. Goebel, C. Kruegel, and E. Kirda, “Automatically generating models for botnet detection,” in *European Symposium on Research in Computer Security*, 2009, pp. 232–249.
- [237] G. Wondracek, P. M. Comparetti, C. Kruegel, E. Kirda, and S. S. S. Anna, “Automatic network protocol analysis,” in *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, vol. 8, 2008, pp. 1–14.
- [238] P. M. Comparetti, G. Wondracek, C. Kruegel, and E. Kirda, “Prospex: Protocol specification extraction,” in *30th IEEE Symposium on Security and Privacy (SP)*, 2009, pp. 110–125.
- [239] C. Y. Cho, D. Babic, P. Poosankam, K. Z. Chen, E. X. Wu, and D. Song, “Mace: Model-inference-assisted concolic exploration for protocol and vulnerability discovery,” in *20th USENIX Security Symposium (USENIX Security)*, vol. 139, 2011.
- [240] M. A. Beddoe, “Network protocol analysis using bioinformatics algorithms,” *Toorcon*, 2004.

- [241] C. Leita, K. Mermoud, and M. Dacier, “Scriptgen: An automated script generation tool for honeyd,” in *21st Annual Computer Security Applications Conference (AC-SAC)*, 2005, 12–pp.
- [242] G. Bossert, F. Guihéry, and G. Hiet, “Towards automated protocol reverse engineering using semantic information,” in *Proceedings of the 9th ACM Symposium on Information, Computer and Communications Security*, 2014, pp. 51–62.
- [243] Y. Wang, Z. Zhang, D. D. Yao, B. Qu, and L. Guo, “Inferring protocol state machine from network traces: A probabilistic approach,” in *International Conference on Applied Cryptography and Network Security*, 2011, pp. 1–18.
- [244] W. Cui, J. Kannan, and H. J. Wang, “Discoverer: Automatic protocol reverse engineering from network traces,” in *16th USENIX Security Symposium (USENIX Security)*, 2007, pp. 1–14.
- [245] D.-F. Feng and R. F. Doolittle, “Progressive sequence alignment as a prerequisite to correct phylogenetic trees,” *Journal of Molecular Evolution*, vol. 25, no. 4, pp. 351–360, 1987.
- [246] Y. Ye, Z. Zhang, F. Wang, X. Zhang, and D. Xu, “Netplier: Probabilistic network protocol reverse engineering from message traces,” in *NDSS*, 2021.
- [247] Y. Shao, J. Ott, Q. Alfred Chen, Z. Qian, and Z. Morley Mao, “Kratos: Discovering Inconsistent Security Policy Enforcement in the Android Framework,” Internet Society, May 2017. DOI: [10.14722/ndss.2016.23046](https://doi.org/10.14722/ndss.2016.23046).
- [248] Y. Aafer, J. Huang, Y. Sun, X. Zhang, N. Li, and C. Tian, “AceDroid: Normalizing Diverse Android Access Control Checks for Inconsistency Detection,” Internet Society, Feb. 2018. DOI: [10.14722/ndss.2018.23121](https://doi.org/10.14722/ndss.2018.23121).
- [249] S. A. Gorski *et al.*, “ACMiner: Extraction and Analysis of Authorization Checks in Android’s Middleware,” Jan. 2019. [Online]. Available: <http://arxiv.org/abs/1901.03603>.
- [250] S. A. Gorski III, S. Thorn, W. Enck, and H. Chen, “{Fred}: Identifying file {re-delegation} in android system services,” in *31st USENIX Security Symposium (USENIX Security 22)*, 2022, pp. 1525–1542.
- [251] H. Zhou, H. Wang, X. Luo, T. Chen, Y. Zhou, and T. Wang, “Uncovering cross-context inconsistent access control enforcement in android,” 2022.

- [252] Z. El-Rewini, Z. Zhang, and Y. Aafer, “Poirot: Probabilistically recommending protections for the android framework,” in *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, 2022, pp. 937–950.
- [253] <https://docs.microsoft.com/en-us/windows/security/threat-protection/intelligence/fileless-threats>.
- [254] <https://www.cybereason.com/blog/fileless-malware>.
- [255] <https://blog.barkly.com/powerpoint-malware-installs-when-users-hover-over-a-link>.
- [256] https://www.trendmicro.com/vinfo/us/threat-encyclopedia/malware/hktml_coinmine.
- [257] <https://gbhackers.com/evil-clone-attack-legitimate-pdf-software>.
- [258] <https://blog.alertlogic.com/10-must-know-2018-cybersecurity-statistics/>.
- [259] <https://padawan.s3.eurecom.fr/about>.
- [260] J. Bergeretti and B. Carré, “Information-flow and data-flow analysis of while-programs,” *ACM Trans. Program. Lang. Syst.*, vol. 7, no. 1, pp. 37–61, 1985.
- [261] J. Ferrante, K. J. Ottenstein, and J. D. Warren, “The program dependence graph and its use in optimization,” *ACM Trans. Program. Lang. Syst.*, vol. 9, no. 3, pp. 319–349, 1987.
- [262] A. Sabelfeld and A. C. Myers, “Language-based information-flow security,” *IEEE Journal on Selected Areas in Communications*, vol. 21, no. 1, pp. 5–19, 2003.
- [263] A. C. Myers, “Jflow: Practical mostly-static information flow control,” in *POPL ’99, Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Antonio, TX, USA, January 20-22, 1999*, 1999, pp. 228–241.
- [264] J. A. Clause, W. Li, and A. Orso, “Dytan: A generic dynamic taint analysis framework,” in *Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2007, London, UK, July 9-12, 2007*, 2007, pp. 196–206.

- [265] J. Newsome and D. X. Song, “Dynamic taint analysis for automatic detection, analysis, and signaturegeneration of exploits on commodity software,” in *Proceedings of the Network and Distributed System Security Symposium, NDSS 2005, San Diego, California, USA*, 2005.
- [266] E. Zhu *et al.*, “Dytaint: The implementation of a novel lightweight 3-state dynamic taint analysis framework for x86 binary programs,” *Computers & Security*, vol. 52, pp. 51–69, 2015.
- [267] K. Olmos and E. Visser, “Composing source-to-source data-flow transformations with rewriting strategies and dependent dynamic rewrite rules,” in *Compiler Construction, 14th International Conference, CC 2005, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2005, Edinburgh, UK, April 4-8, 2005, Proceedings*, 2005, pp. 204–220.
- [268] J. Bell, G. E. Kaiser, E. Melski, and M. Dattatreya, “Efficient dependency detection for safe java test acceleration,” in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, Bergamo, Italy, August 30 - September 4, 2015*, 2015, pp. 770–781.
- [269] V. K. Palepu, G. (Xu, and J. A. Jones, “Improving efficiency of dynamic analysis with dynamic dependence summaries,” in *2013 28th IEEE/ACM International Conference on Automated Software Engineering, ASE 2013, Silicon Valley, CA, USA, November 11-15, 2013*, 2013, pp. 59–69.
- [270] B. Steensgaard, “Points-to analysis in almost linear time,” in *Conference Record of POPL ’96: The 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Papers Presented at the Symposium, St. Petersburg Beach, Florida, USA, January 21-24, 1996*, 1996, pp. 32–41.
- [271] D. Liang and M. J. Harrold, “Efficient points-to analysis for whole-program analysis,” in *Software Engineering - ESEC/FSE’99, 7th European Software Engineering Conference, Held Jointly with the 7th ACM SIGSOFT Symposium on the Foundations of Software Engineering, Toulouse, France, September 1999, Proceedings*, 1999, pp. 199–215.
- [272] M. Emami, R. Ghiya, and L. J. Hendren, “Context-sensitive interprocedural points-to analysis in the presence of function pointers,” in *Proceedings of the ACM SIGPLAN’94 Conference on Programming Language Design and Implementation (PLDI), Orlando, Florida, USA, June 20-24, 1994*, 1994, pp. 242–256.

- [273] A. Deutsch, “Interprocedural may-alias analysis for pointers: Beyond k -limiting,” in *Proceedings of the ACM SIGPLAN’94 Conference on Programming Language Design and Implementation (PLDI), Orlando, Florida, USA, June 20-24, 1994*, 1994, pp. 230–241.
- [274] V. Kahlon, “Bootstrapping: A technique for scalable flow and context-sensitive pointer alias analysis,” in *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation, Tucson, AZ, USA, June 7-13, 2008*, 2008, pp. 249–259.
- [275] X. Zheng and R. Rugina, “Demand-driven alias analysis for C,” in *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008, San Francisco, California, USA, January 7-12, 2008*, 2008, pp. 197–208.
- [276] M. Hirzel, D. von, Dincklage, A. Diwan, and M. Hind, “Fast online pointer analysis,” *ACM Trans. Program. Lang. Syst.*, vol. 29, no. 2, p. 11, 2007.
- [277] R. Thiessen and O. Lhoták, “Context transformations for pointer analysis,” in *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017*, 2017, pp. 263–277.
- [278] G. Xu and A. Rountev, “Merging equivalent contexts for scalable heap-cloning-based context-sensitive points-to analysis,” in *Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2008, Seattle, WA, USA, July 20-24, 2008*, 2008, pp. 225–236.
- [279] I. Dillig, T. Dillig, and A. Aiken, “Sound, complete and scalable path-sensitive analysis,” in *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation, Tucson, AZ, USA, June 7-13, 2008*, 2008, pp. 270–280.
- [280] M. Das, S. Lerner, and M. Seigle, “ESP: path-sensitive program verification in polynomial time,” in *Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Berlin, Germany, June 17-19, 2002*, 2002, pp. 57–68.
- [281] S. Gulwani and G. C. Necula, “Precise interprocedural analysis using random interpretation,” in *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2005, Long Beach, California, USA, January 12-14, 2005*, 2005, pp. 324–337.

- [282] S. Gulwani and G. C. Necula, “Global value numbering using random interpretation,” in *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2004, Venice, Italy, January 14-16, 2004*, 2004, pp. 342–352.
- [283] S. Gulwani and G. C. Necula, “Discovering affine equalities using random interpretation,” in *Conference Record of POPL 2003: The 30th SIGPLAN-SIGACT Symposium on Principles of Programming Languages, New Orleans, Louisiana, USA, January 15-17, 2003*, 2003, pp. 74–84.
- [284] B. D. Sutter, B. D. Bus, K. D. Bosschere, P. Keyngnaert, and B. Demoen, “On the static analysis of indirect control transfers in binaries,” in *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications, PDPTA 2000, June 24-29, 2000, Las Vegas, Nevada, USA, 2000*.
- [285] H. Theiling, “Extracting safe and precise control flow from binaries,” in *7th International Workshop on Real-Time Computing and Applications Symposium (RTCSA 2000), 12-14 December 2000, Cheju Island, South Korea, 2000*, pp. 23–30.
- [286] C. Kolbitsch, T. Holz, C. Kruegel, and E. Kirda, “Inspector gadget: Automated extraction of proprietary gadgets from malware binaries,” in *31st IEEE Symposium on Security and Privacy, S&P 2010, 16-19 May 2010, Berkeley/Oakland, California, USA, 2010*, pp. 29–44.
- [287] M. H. Nguyen, T. B. Nguyen, T. T. Quan, and M. Ogawa, “A hybrid approach for control flow graph construction from binary code,” in *20th Asia-Pacific Software Engineering Conference, APSEC 2013, Ratchathewi, Bangkok, Thailand, December 2-5, 2013 - Volume 2*, 2013, pp. 159–164.
- [288] W. He *et al.*, “Rethinking access control and authentication for the home internet of things (iot),” in *27th {USENIX} Security Symposium ({USENIX} Security 18)*, 2018, pp. 255–272.
- [289] E. Gustafson *et al.*, “Toward the analysis of embedded firmware through automated re-hosting,” in *22nd International Symposium on Research in Attacks, Intrusions and Defenses ({RAID} 2019)*, 2019, pp. 135–150.
- [290] G. Hernandez, F. Fowze, D. Tian, T. Yavuz, and K. R. Butler, “Firmusb: Vetting usb device firmware using domain informed symbolic execution,” in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017, pp. 2245–2262.

- [291] K. Jansen, M. Schäfer, D. Moser, V. Lenders, C. Pöpper, and J. Schmitt, “Crowd-gps-sec: Leveraging crowdsourcing to detect and localize gps spoofing attacks,” in *2018 IEEE Symposium on Security and Privacy (SP)*, IEEE, 2018, pp. 1018–1031.
- [292] D. Freed *et al.*, ““ is my phone hacked?” analyzing clinical computer security interventions with survivors of intimate partner violence,” *Proceedings of the ACM on Human-Computer Interaction*, vol. 3, no. CSCW, pp. 1–24, 2019.
- [293] H. Cho *et al.*, “Prime+Count: Novel Cross-world Covert Channels on ARM TrustZone,” in *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*, Dec. 2018.
- [294] P. Kocher *et al.*, “Spectre attacks: Exploiting speculative execution,” in *2019 IEEE Symposium on Security and Privacy (SP)*, IEEE, 2019, pp. 1–19.
- [295] A. Reina, A. Fattori, F. Pagani, L. Cavallaro, and D. Bruschi, “When hardware meets software: A bulletproof solution to forensic memory acquisition,” in *Proceedings of the 28th annual computer security applications conference*, 2012, pp. 79–88.
- [296] M. Schwarz and A. Fogh, “Drama: How your dram becomes a security problem,” 2016.
- [297] L. Demetrio, B. Biggio, G. Lagorio, F. Roli, and A. Armando, “Explaining vulnerabilities of deep learning to adversarial malware binaries,” *arXiv preprint arXiv:1901.03583*, 2019.
- [298] M. Schwarz *et al.*, “Automated detection, exploitation, and elimination of double-fetch bugs using modern cpu features,” in *Proceedings of the 2018 on Asia Conference on Computer and Communications Security*, 2018, pp. 587–600.
- [299] H. Lee, C. Song, and B. B. Kang, “Lord of the x86 rings: A portable user mode privilege separation architecture on x86,” in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018, pp. 1441–1454.
- [300] X. Xu, C. Liu, Q. Feng, H. Yin, L. Song, and D. Song, “Neural network-based graph embedding for cross-platform binary code similarity detection,” in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*, 2017, pp. 363–376.

- [301] W. Guo, D. Mu, M. Du, X. Xing, and D. Song., “Deepvsa: Facilitating value-set analysis with deep learning for postmortem program analysis,” in *28th USENIX Security Symposium, USENIX Security 2019*, 2019.
- [302] J. Geldenhuys, M. B. Dwyer, and W. Visser, “Probabilistic symbolic execution,” in *International Symposium on Software Testing and Analysis, ISSTA 2012, Minneapolis, MN, USA, July 15-20, 2012*, 2012, pp. 166–176.
- [303] M. Borges, A. Filieri, M. d’Amorim, and C. S. Pasareanu, “Iterative distribution-aware sampling for probabilistic symbolic execution,” in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, Bergamo, Italy, August 30 - September 4, 2015*, 2015, pp. 866–877.
- [304] M. Z. Kwiatkowska, G. Norman, and D. Parker, “PRISM 4.0: Verification of probabilistic real-time systems,” in *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*, 2011, pp. 585–591.
- [305] A. Filieri, C. Ghezzi, and G. Tamburrelli, “Run-time efficient probabilistic model checking,” in *Proceedings of the 33rd International Conference on Software Engineering, ICSE 2011, Waikiki, Honolulu , HI, USA, May 21-28, 2011*, 2011, pp. 341–350.
- [306] A. F. Donaldson, A. Miller, and D. Parker, “Language-level symmetry reduction for probabilistic model checking,” in *QEST 2009, Sixth International Conference on the Quantitative Evaluation of Systems, Budapest, Hungary, 13-16 September 2009*, 2009, pp. 289–298.
- [307] Y. Cai, J. Zhang, L. Cao, and J. Liu, “A deployable sampling strategy for data race detection,” in *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016, Seattle, WA, USA, November 13-18, 2016*, 2016, pp. 810–821.
- [308] M. Hauswirth and T. M. Chilimbi, “Low-overhead memory leak detection using adaptive statistical profiling,” in *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2004, Boston, MA, USA, October 7-13, 2004*, 2004, pp. 156–164.
- [309] Y. Zhong and W. Chang, “Sampling-based program locality approximation,” in *Proceedings of the 7th International Symposium on Memory Management, ISMM 2008, Tucson, AZ, USA, June 7-8, 2008*, 2008, pp. 91–100.

- [310] N. Toronto, J. McCarthy, and D. V. Horn, “Running probabilistic programs backwards,” in *Programming Languages and Systems - 24th European Symposium on Programming, ESOP 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings*, 2015, pp. 53–79.
- [311] A. Avizienis, “The n-version approach to fault-tolerant software,” *IEEE Transactions on software engineering*, no. 12, pp. 1491–1501, 1985.
- [312] E. D. Berger and B. G. Zorn, “Diehard: Probabilistic memory safety for unsafe languages,” in *PLDI*, M. I. Schwartzbach and T. Ball, Eds., 2006, pp. 158–168.
- [313] J. Xu, B. Randell, A. Romanovsky, C. M. Rubira, R. J. Stroud, and Z. Wu, “Fault tolerance in concurrent object-oriented software through coordinated error recovery,” in *Twenty-Fifth International Symposium on Fault-Tolerant Computing. Digest of Papers*, IEEE, 1995, pp. 499–508.
- [314] J. Xu, A. Romanovsky, and B. Randell, “Concurrent exception handling and resolution in distributed object systems,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 11, no. 10, pp. 1019–1032, 2000.
- [315] J. Oberheide, E. Cooke, and F. Jahanian, “Clouday: N-version antivirus in the network cloud,” in *USENIX Security*, 2008, pp. 91–106.
- [316] A. Carzaniga, A. Gorla, A. Mattavelli, N. Perino, and M. Pezzè, “Automatic recovery from runtime failures,” in *ICSE*, 2013.
- [317] S. Nagy and M. Hicks, “Full-speed fuzzing: Reducing fuzzing overhead through coverage-guided tracing,” in *SP*, 2019, pp. 787–802.